UNIVERSITETET I OSLO
Institutt for informatikk

Bruk av mobile agenter til vedlikehold i distribuerte systemer

Per Thomas Jahr

Hovedfagsoppgave

1. november 1999

Bruk av mobile agenter til vedlikehold i distribuerte systemer

Hovedfagsoppgave ved Institutt for informatikk

Per Thomas Jahr

1. november 1999

Denne rapporten er produsert ved hjelp av programmet LaTeX og en rekke tilleggspakker. Typesnittet i denne rapporten er Palatino 11 punkt. Bibliografien er satt sammen med programmet BibTeX. All tekst er skrevet i `GNU Emacs`, og figurene er tegnet i FrameMaker eller behandlet med `xv`.
Omslag og innbinding: Strandberg & Nilsen Grafisk A/S

Sammendrag

Denne oppgaven undersøker hvordan mobile agenter kan brukes til vedlikehold i distribuerte systemer. Vedlikehold av programvare forbindes ofte med feilretting, men undersøkelser har vist at en stor del av vedlikeholdet er videreutvikling. En utfordring er derfor å finne fram til metoder som gjør det lettere å endre på programvare.

Med utgangspunkt i dette undersøker jeg hvordan mobile agenter kan brukes som en teknikk for å endre programmer. Mobile agenter er små programmer som kan flyttes rundt i et nettverk for å utføre ulike oppgaver. En av fordelene er at en agent kan inneholde kode som utvider funksjonaliteten eller retter feil i et program. Mobile agenter kan brukes som en teknikk for å distribuere og installere kode.

Mobile agenter kan også brukes i programvare som har store krav til fleksibilitet. Eksempel på dette er programmer som brukes i mobile enheter som håndholdte maskiner. Mobile agenter kan støtte adaptivitet i denne typen programvare.

I denne oppgaven presenterer jeg en enkel prototype som er et eksempel på hvordan en distribuert applikasjon kan oppgraderes ved hjelp av en mobil agent. Eksemplet viser hvordan mobile agenter kan brukes som en teknikk for fleksibelt vedlikehold og tilpasning av programvare.

Graden av vedlikehold kan ofte spores tilbake til begynnelsen av et utviklingsprosjekt. Grunnlaget for endringer og videreutvikling bestemmes av arkitekturen til programmet. I denne oppgaven presenterer jeg en modell for hvordan vedlikehold med mobile agenter kan brukes fra første fase i et prosjekt. Formålet med modellen er å undersøke hvordan mobile agenter kan danne grunnlaget for en arkitektur som gjør programvare mer åpen for endringer.

Oppgaven avslutter med å diskutere forskjellige utfordringer ved det å bruke mobile agenter til vedlikehold.

Forord

Dette er min hovedfagsoppgave til cand.scient graden innen studieretningen kommunikasjonssystemer ved Institutt for informatikk. Under arbeidet med denne oppgaven er det mange person som fortjener en stor takk. Først og fremst vil jeg takke Steinar Kristoffersen fra Norsk Regnesentral (NR) for bra veiledning gjennom to år. I løpet av denne perioden har jeg fått mange gode idéer og forslag til hvordan oppgaven burde skrives. Steinar har også gitt meg muligheten til å delta på ulike prosjekter. Et resultat av dette er prototypen som presenteres i denne oppgaven. En stor takk til Arne-Kristian Groven og Thor Kristoffersen fra NR som var med på å designe prototypen.

Gjennom to år har jeg samarbeidet med mange medstudenter. Spesielt vil jeg takke Sverre Steensen som også har skrevet en hovedoppgave om mobile agenter. Diskusjonene vi har hatt om mobile agenter har vært svært nyttige. Jeg har også samarbeidet med Espen Dåsvand og Bjørn Aadland på to mindre prosjekter. Disse prosjektene har vært nyttige for å utforme denne oppgaven. I tillegg har de gitt programmeringserfaring.

I løpet av de to årene jeg har holdt på med denne oppgaven, har jeg hatt stor nytte av tilbakemeldinger fra medstudenter. Gjennom en mer eller mindre organisert "lesering" har jeg og andre fått kommentarer på det vi har skrevet. Takk til Eirik Maus, Ole Henrik Dahle, Rune Schumann, Eirik Torske og Knut-Olav Traa for gode tilbakemeldinger.

Under siste del av arbeidet med denne oppgaven vil jeg takke Torgeir Fritsvold og Igor Rafienko for gjennomlesing og forslag til de siste forbedringene av oppgaven. En stor takk går også til min gode venn Tore Sæthermoen for korrekturlesing av oppgaven.

Til slutt en stor takk til min samboer Bente som gjennom disse to årene har holdt ut med meg og oppgaven min. Uten støtte og hjelp fra deg ville det ikke ha vært lett å gjennomføre denne oppgaven.

Oslo, 1. november 1999

Per Thomas Jahr

Innhold

Kapittel 1

Innledning

Denne oppgaven handler om hvordan mobile agenter kan brukes til å utføre vedlikehold i distribuerte systemer. Mobile agenter er små programmer som kan flyttes rundt i et nettverk for å utføre forskjellige oppgaver. En agent kan brukes til å distribuere, installere og oppgradere programmer i et nettverk. Et nettverk av datamaskiner, som på en eller annen måte samarbeider, kalles gjerne for et distribuert system.

Vedlikehold av programvare kan utgjøre mellom 40 til 70 prosent av de totale utgiftene til en applikasjon [TG96, side 2]. Dette omfatter ikke bare retting av feil, men også det å legge til ny funksjonalitet i et program. Det som skiller vedlikehold fra nyutvikling, er at vedlikehold foregår etter at systemet er tatt i bruk av kunden.

Et eksempel på vedlikehold er år 2000-problemet (Y2K[1]-problemet). Kort fortalt går Y2K-problemet ut på at en del program- og maskinvare ikke klarer overgangen fra år 1999 til år 2000. På nyttårsaften vil de to siste sifrene skifte fra 99 til 00. En del beregninger som tar utgangspunkt i disse sifrene vil da gi feil resultat.

Y2K-problemet er et vedlikeholdsproblem fordi det er nødvendig å forandre på systemer som allerede er installert. De som skal utføre vedlikeholdet må derfor ta hensyn til at systemet er i daglig bruk. For å sikre stabil drift kan det være nødvendig å sette opp en kopi av systemet. Vedlikehold kan også føre til at flere feil blir innført i systemet. Dette er ofte tilfellet i store systemer hvor det er vanskelig å ha full oversikt.

I denne oppgaven undersøker jeg hvordan mobile agenter kan brukes til å endre programmer. Dette gjelder både distribusjon, installering, feilretting og ulike tilpasninger av programmer. Et scenario for bruk av mobile agenter finnes i store bedrift med mange datamaskiner koblet sammen via et nettverk. For å oppgradere programvare på alle maskinene, kan en eller flere agenter sendes ut i nettverket. Agentene vil forflytte seg fra maskin til

[1] Y2K er forkortelse for *Year 2 Kilo* - eller år 2000.

maskin. For hver maskin kan agentene oppgradere programvaren. På denne måten kan agenter for eksempel brukes til å rette opp Y2K-feil i programvaren. Agentene kan også programmeres slik at de tar spesielle hensyn til forskjeller i operativsystemer, brukeroppsett og maskinvare.

En forutsetning for dette scenarioet er at den distribuerte applikasjonen har støtte for mobile agenter. Dette vil si at hvert program må kunne ta i mot agenter samt tilby en omgivelse de kan eksekvere i. I denne oppgaven undersøker jeg hvordan mobile agenter kan brukes i applikasjoner for å utføre vedlikehold, og hvordan distribuerte applikasjoner må endres for at dette skal være mulig.

1.1 Bakgrunn

Mer og mer i samfunnet styres av datateknologi. I det offentlige er dette nødvendig for å administrere og styre store mengder informasjon på en effektiv måte. I det private brukes informasjonsteknologi både til underholdning og arbeid.

Denne økte bruken av datateknologi skaper også et behov for vedlikehold og videreutvikling av programvare. Det er mange grunner til dette. En av de fremste grunnene er den raske utviklingen innen program- og maskinvare. Programvare må endres slik at de kan utnytte fordelene ved en ny prosessor eller et nytt program.

En annen motivasjon for å utføre vedlikehold er at et fåtall av programmene som utvikles i dag er uten feil. En årsak er at programmer over en viss størrelse er såpass komplekse at feil alltid oppstår [Jr.87]. Konsekvensene av feil i et program kan være alvorlige, men dette avhenger selvfølgelig av type program. Ingen programmer bør i utgangspunktet inneholde feil, men i noen programmer er det kritisk med feil (for eksempel i et system for flykontroll).

I de siste årene har vi sett hvordan Internett har utviklet seg fra et forskningsprosjekt til et daglig verktøy og underholdningsmedium. I Norge har 41 prosent av befolkningen tilgang til Internett hjemme, på jobben eller via skole[2]. Internett er kanskje mest kjent for tjenestene *World Wide Web* og e-post, men tjenester for å laste ned programvare brukes også mye.

Bruken av Internett som en distribusjonskanal for programmer og oppdateringer er ikke ny, men vanligvis er installering overlatt til bruker. Ved hjelp av programmer for nedlasting av filer kan en bruker laste ned et hvilket

[2]Tallene er hentet fra en artikkel i Aftenposten 11.november 1998 - "Nedgang i daglig bruk av Internett - Oslofolk på nett-toppen". Tallene baserer seg på en undersøkelse foretatt av Norsk Gallup (`http://www.gallup.no` - 10.oktober 1999).

som helst program, men brukeren må selv sørge for å pakke ut og installere programmet. Dette kan være problematisk fordi programmet kanskje er avhengig av andre programmer eller programbiblioteker for å fungere.

I tillegg øker kravene til kvalitet på programvare. Kvalitetskravene kan deles inn i mange kategorier [Som96, side 612]. Et generelt krav er at et program bør være lett å bruke (brukervennlig) samtidig som det bør være effektivt. Et annet krav kan være at et program skal være lett å endre. Dette er en fordel hvis for eksempel en bedrift vil innføre en ny tjeneste i sitt datasystem, eller hvis programvaren må endres på grunn av lover og regler. Tradisjonelle programmer er stort sett statiske og fanger ikke opp forandringer i omgivelsene. Hvis programvaren skal forandres må man enten utføre vedlikehold eller anskaffe et nytt program.

Vi ser også at datautstyr tar mindre plass samtidig som de får nye bruksområder. Et eksempel på dette er håndholdte maskiner (PDA[3]) eller nye mobiltelefoner. Disse enhetene har andre krav til programvare fordi strøm, prosessorkapasitet, båndbredde og skjermstørrelse er begrenset. En bruker kan ikke uten videre benytte samme e-postprogram på en PC og en mobiltelefon. Utfra en brukers synspunkt kan det være ønskelig å bruke det samme programmet på begge enhetene. For at dette skal være mulig må programvaren tilpasse seg ulike situasjoner. Dette kalles gjerne *adaptivitet*.

Alle disse faktorene gjør at vi trenger nye teknikker som støtter endring av programvare for å:

1. dynamisk legge til eller erstatte kode i applikasjoner i et distribuert system,
2. gjøre applikasjoner mer tilpasningsdyktige ved at de kan laste ned kode som er tilpasset forskjellige situasjoner, og
3. klargjøre programvare for endringer allerede fra første fase i utviklingsprosessen.

I denne oppgaven vil jeg utforske hvordan mobile agenter kan brukes til å løse disse tre oppgavene.

1.1.1 Andre teknikker for vedlikehold

Det å bruke Internett til å spre programvare og utføre vedlikehold kan gjøres på forskjellige måter. FTP[4] er et av de første eksemplene på en protokoll som kan brukes til å laste ned programmer via nettet. Ulempen er at installering av programvaren ofte er overlatt til brukeren.

[3] PDA står for *Personal Digital Assistent* eller personlig digital assistent.
[4] FTP står for *File Transfer Protocol*.

I spill og operativsystemer brukes såkalte *patcher*. En *patch* er et program for å oppgradere eller rette feil direkte på binærfiler eller objektfiler. Dette er ofte en nødløsning for å rette opp en eller flere kritiske feil. En *patch* behøver derfor ikke å være den beste løsningen på et problem. Andre og bedre måter å løse feilene på er ofte implementert i senere versjoner av programvaren.

Oppgaven med å starte en *patch* er overlatt til brukeren, eller et spesielt program kan brukes til å administrere ulike *patcher*. Dette kan være nødvendig hvis mange *patcher* må installeres i en spesiell rekkefølge.

Castanet fra Marimba[5] er et kommersielt produkt som støtter automatisk spredning av programvare og oppdateringer innenfor et nettverk. Castanet bruker en teknikk som kalles *differential update* eller *tuning*. Denne teknikken oppdaterer de filene som er forandret siden sist. På denne måten slipper man å laste ned hele programpakker.

Andre teknikker som for eksempel *plug-ins* fra Netscape[6] eller *AutoUpdate*[7] i RealPlayer G2 er eksempler på teknikker for å automatisk utvide funksjonaliteten. Støtte for nye protokoller, standarder eller filformater kan automatisk lastes ned fra nettet.

Problemet med disse teknikkene er at de er svært forskjellige og støtter ulike bruksområder. Noen av teknikkene har støtte for automatisk distribusjon og installering av programvare, mens andre bare tilbyr distribusjon. Teknikkene varierer også med hensyn til om oppgraderingen er planlagt eller ikke. En *patch* er for eksempel en ikke-planlagt endring, mens *plug-ins* og *AutoUpdate* er eksempler på at mekanismer for vedlikehold er integrert i programvaren. Castanet er en tredje variant som er spesielt rettet mot administrasjon og drift av programvare i et nettverk. Det er derfor ikke nødvendig å programmere inn støtte for Castanet i et program.

Få av teknikkene støtter dynamiske endringer. Dette vil si at endringene er tilgjengelig for en bruker uten at applikasjonen må startes på nytt. Dette kan være nyttig for å understøtte adaptivitet i blant annet trådløse nett. En håndholdt enhet kan for eksempel bli flyttet mellom forskjellige soner eller celler i et nettverk. For at et program skal fungere i de forskjellige sonene, kan det være nødvendig å laste ned kode som programmet tar i bruk mens det eksekverer.

Mobile agenter *kan* være et nytt prinsipp for distribusjon, oppgradering, vedlikehold og adapsjon av forskjellige typer programvare. Bruk av mobile agenter i designfasen kan også være en måte å klargjøre applikasjoner for

[5] `http://www.marimba.com` (31. oktober 1999)

[6] `http://www.netscape.com` (31.oktober 1999)

[7] Mer om *AutoUpdate* finnes på `http://www.real.com/devzone/library/whitepapers/autoupdate.h` (21. oktober 1999).

fremtidige endringer. I dag er det vanlig å identifisere klasser av objekter. Dette er en måte å strukturere et system på. Mobile agenter kan kanskje brukes til å gruppere flyttbare eller utskiftbare deler av en applikasjon.

1.2 Problemstilling

Tema for denne oppgaven er å vurdere hvordan mobile agenter kan brukes til vedlikehold i distribuerte systemer. For å undersøke dette har jeg satt opp en overordnet problemstilling og delt denne opp i flere delproblemer. Problemstillingen jeg har tatt utgangspunkt i er:

> *Hvordan egner mobile agenter seg til å utføre vedlikehold i distribuerte systemer?*

Mobile agenter er sentralt i denne oppgaven. En mobil agent er et lite program som kan flyttes rundt i et nettverk for å utføre ulike oppgaver. Begrepet mobil agent er såpass nytt at det er interessant å finne muligheter og begrensninger ved denne teknologien. Mobile agenter handler om å flytte kode til et hensiktsmessig sted for å utnytte lokal kommunikasjon eller for å få tilgang til lokale tjenester.

Vedlikehold er et samlebegrep på endringer som blir utført på et system etter at det er tatt i bruk. I denne oppgaven er det viktig å finne fram til hvilke typer vedlikehold som finnes og hvilke utfordringer disse gir. Videre er det interessant å finne ut hva slags vedlikehold mobile agenter kan støtte.

Distribuerte systemer er det tredje stikkordet for denne oppgaven. Enkelt forklart er et distribuert system en samling datamaskiner som kommuniserer ved hjelp av et nettverk. Distribuerte systemer har en del egenskaper som kompliserer programvare. Eksempler på dette er datasikkerhet og heterogenitet. Mobile agenter utnytter tjenestene i et distribuert system, og utfordringene i disse systemene kan også gjelde for mobile agenter. Det er derfor viktig å finne fram til konsekvensene av å bruke mobile agenter.

Den overordnede problemstillingen har jeg delt opp disse delproblemene:

- *Hvordan kan mobile agenter støtte vedlikehold i et distribuert system?*
  Hensikten er å finne fram til de grunnleggende teknikkene for å frakte kode over nettet samt installere kode i en applikasjon. Det er også viktig å kategorisere vedlikehold for å finne ut hvilke typer vedlikehold mobile agenter kan utføre.

- *Hvordan kan mobile agenter støtte adaptive applikasjoner?*
  Mobile agenter kan kanskje brukes til dynamisk oppgradering av en

applikasjon. Det er derfor interessant å finne fram til fordeler og ulemper ved å bruke mobile agenter til å støtte adaptive applikasjoner.

- *Hvordan kan mobile agenter gjøre applikasjoner mer åpne for endringer?* Mobile agenter kan også betraktes som en måte å organisere kommunikasjonen i et distribuert system. Bruk av mobile agenter bør derfor ses i sammenheng med teknikkene vi bruker under design av programvare. Kanskje kan en applikasjon designet med mobile agenter være mer åpen for endringer?

1.2.1 Presisering og avgrensing av oppgaven

I denne oppgaven ser jeg på hvordan mobile agenter kan brukes som en teknikk for forskjellige typer vedlikehold. Vedlikehold er bare en av mange foreslåtte bruksområder for agenter. For eksempel er de egnet til å søke etter informasjon i store nettverk [FPV98]. Dette at mobile agenter kan brukes som en løsning innen flere områder, blir trukket fram som en av de store fordelene ved agenter [CHK97] (mer om dette på side 29). For å avgrense omfanget av denne oppgaven kommer jeg ikke til å se nærmere på hvordan agenter kan utføre andre oppgaver i kombinasjon med en vedlikeholdsoppgave.

I følge [Som96] er vedlikehold en omfattende prosess som består av alt fra innmelding av feil, konsekvensanalyse og til slutt implementering. Når det gjelder vedlikehold av programvare vil denne oppgaven fokusere mest på de tekniske sidene ved vedlikehold. Denne oppgaven ser for eksempel ikke nærmere på hvordan dokumentasjonen til et system skal vedlikeholdes eller oppgraderes. I denne oppgaven ser jeg heller ikke nærmere på ulike prosesser i forbindelse med rutiner for kvalitet eller forbedring av kvalitet i en organisasjon.

Sikkerhet i forbindelse med mobile agenter er et viktig forskningsområde. Mobile agenter består av kode som kan eksekvere på forskjellige maskiner i et nettverk. Det er derfor viktig at så vel agent som maskin er beskyttet mot virus og ondsinnede agenter. Sikkerhet er et omfattende område og er i liten grad tatt med i denne oppgaven.

For å endre på et program er det nødvendig å forstå koden til programmet. Ved bruk av mobile agenter kan oppgaven med å forstå koden til en applikasjon bli enda vanskeligere enn i programmer som ikke bruker mobile agenter. Dette er på grunn av at koden kan være spredt utover mange maskiner. Denne oppgaven vil i liten grad komme inn på teknikker for å strukturere kode slik at den blir lettere å forstå.

Før en mobil agent blir sendt ut i et nettverk, er det viktig å instruere agenten med regler for hva den skal gjøre. Dette kan gjøres på forskjellige måter.

I denne oppgaven er reglene for hva agenten skal gjøre programmert inn i agenten. Mer fleksible løsninger ville ha tillatt en bruker å instruere agenten. I kapittel 7 blir en teknikk for instruksjon av agenter diskutert. Utover dette inneholder ikke denne oppgaven mer om hvordan interaksjonen mellom en bruker og en mobil agent kan foregå.

1.3 Metoder

Metodene jeg har brukt for å undersøke de forskjellige problemene i denne oppgaven har vært litteraturstudier og eksperimentering med en prototype som bruker mobile agenter.

Litteraturstudiene har vært konsentrert rundt mobile agenter, distribuerte systemer og vedlikehold. Mobile agenter er et forholdsvis nytt begrep og det brukes i forskjellige sammenhenger. Det har derfor vært viktig å finne en definisjon som det er bred enighet om.

Jeg har også sett nærmere på hva distribuerte systemer er og hvilke spesielle egenskaper disse har. Selv om en god del av problemene blir skjult av såkalt mellomvare[8], er det allikevel viktig å vite hva som er hovedutfordringene når det gjelder å utvikle et distribuert system. Noen av disse utfordringene kan også gjelde for mobile agenter.

Vedlikehold er kjernen i denne oppgaven. For å forstå hva vedlikehold er, har jeg studert hvorfor vedlikehold oppstår, hvilke typer vedlikehold som finnes og hva som er problematisk med vedlikehold. Videre har jeg sett på hva som kan gjøres for å bedre vedlikeholdbarheten i programvare.

For å undersøke problemstillingen har jeg programmert et eksempel på en distribuert applikasjon. Denne applikasjonen har fått navnet MORP (*Multi-Organisation Resource Planner*). MORP er et klient/tjener system som er tenkt brukt innenfor og mellom samarbeidende bedrifter. Programmet tilbyr en ansatt å reservere møterom. I denne prototypen brukes mobile agenter til å legge til nye funksjoner i klientdelen av MORP.

Det er to formål med prototypen. Foruten å være et praktisk eksempel, bruker jeg konseptene ved MORP til å vurdere hvordan mobile agenter egner seg til vedlikehold. Prototypen er et lite eksempel og viser dermed ikke alle fordeler eller ulemper et slikt system har. Jeg bruker derfor teknikkene fra MORP og prøver å generalisere disse slik at de også passer for andre systemer.

De generelle konseptene fra prototypen oppsummerer jeg i en vedlikeholdsmodell. Denne modellen forklarer begreper og funksjoner som brukes

[8] Mellomvare er forklart nærmere på side 35.

for å beskrive vedlikehold med mobile agenter. Hensikten med modellen er å ha et rammeverk for å kunne diskutere konseptene og utfordringer ved teknikken.

1.4 Viktige resultater

Denne oppgaven trekker fram at vedlikehold kan utføres på nodene i et nettverk der hvor programvaren er installert. Fordelen er at agenten kan programmeres til å ta hensyn til forskjellige faktorer som plattform, brukerpreferanser, maskinvare eller andre forhold i for eksempel nettverket.

Et resultat av oppgaven er også at agenter kan brukes til å støtte adaptive applikasjoner. Dette betyr at en applikasjon dynamisk kan tilpasse sin funksjonalitet i forhold til omgivelsene. Vedlikehold under kjøring krever at programmeringsspråk og kjøretidssystem støtter dynamisk lasting av klasser, pakker eller funksjoner. Dette er for eksempel støttet i programmeringsspråket Java [GJS96]. Problemet i Java er at man ikke kan kontrollere direkte hvilke klasser som skal lastes inn eller fjernes fra den virtuelle maskinen. Dette gjør det komplisert å skifte ut klasser.

I denne oppgaven trekker jeg fram en mulig måte å kategorisere endringer på i distribuerte applikasjoner. Dette er viktig for å finne ut av hvordan vedlikehold i en applikasjon skal foregå. Denne kategoriseringen tar hensyn til når endringen skjer, om endringen er permanent og hvor stort omfang endringen har.

Denne oppgaven presenterer en vedlikeholdsmodell som er et generelt rammeverk for hvordan endringer kan utføres ved hjelp av mobile agenter. Modellen definerer begreper som noder, moduler og mobile agenter. Ved hjelp av begrepene fra modellen kan en vedlikeholdsoppgave deles opp i mindre og kanskje mer håndterlige deler.

1.4.1 Hva kan resultatene brukes til?

Resultatene fra denne oppgaven kan brukes som grunnlag for videre forskning på dynamisk endring av programvare. Vedlikeholdsmodellen sammen med diskusjonen i kapittel 7 har pekt på en del utfordringer. Disse kan være utgangspunktet for videre forskning (forslag til videre arbeid er nevnt på side 90).

Mobile agenter er egnet til adapsjon av programvare. Teknikken i denne oppgaven kan derfor brukes til å utvikle fleksible og brukervennlige applikasjoner. Dette gjelder spesielt i programvare som er ment for flere plattformer. I telekommunikasjon kan mobile agenter brukes til å tilpasse pro-

gramvaren slik at inneholdet i et program (for eksempel et program for e-post) lar seg transportere mellom forskjellige enheter som for eksempel en PC og en mobiltelefon.

Agenter kan også brukes til å utvide tjenestetilbudet i forskjellige nettverk. I GSM[9] nettet kan mobile agenter tilby en bruker ulike funksjoner avhengig av omgivelsene brukeren er i.

Mobile agenter kan også brukes som en grunnleggende arkitektur for distribuerte systemer. Ved hjelp av vedlikeholdsmodellen kan man designe og programmere applikasjoner som er klargjort for ulike typer endringer. Vedlikeholdsmodellen kan da brukes som et forslag til hvordan arkitekturen bør være.

1.5 Presentasjon av oppgaven

Oppbygningen til denne oppgaven er som følger:

- Kapittel 2 ser på sammenhengen mellom evolusjon og vedlikehold av programvare. Dataprogrammer har sin egen dynamikk som gjør det vanskelig å utvikle og vedlikeholde dem. Dette gjelder spesielt store systemer. For å organisere prosessen med å utvikle et system ser jeg på forskjellige systemutviklingsmodeller og hvilken rolle vedlikehold har i disse modellene. Kapitlet avslutter med å trekke fram noen teknikker for å bedre vedlikeholdbarheten i en applikasjon.

- Kapittel 3 presenterer mobile agenter. Mobile agenter er et forholdsvis nytt begrep, så jeg vil se nærmere på bakgrunnen for mobile agenter og på forskjellige definisjoner. Videre ser kapitlet på hvordan mobile agenter er implementert. Et system eller en plattform for å håndtere mobile agenter blir i denne oppgaven kalt for et *agentsystem*. Kapitlet avslutter med å se på det vi tradisjonelt forbinder med distribuerte systemer opp mot mobile agenter.

- I kapittel 4 beskrives prototypen som har fått navnet MORP. Dette kapitlet viser hvordan prototypen virker og hvordan arkitekturen er. Videre brukes prototypen til å demonstrere hvordan en mobil agent kan legge til en ny funksjon i MORP.

- Kapittel 5 evaluerer prototypen MORP. Evalueringen ser både på utviklingsprosessen og prototypen i seg selv. Dette kapitlet fokuserer på fordeler og ulemper ved prototypen.

[9] GSM står for *Global System for Mobile communications.*

- Kapittel 6 generaliserer konseptene fra prototypen. Kapitlet presenterer en vedlikeholdsmodell. Hensikten med modellen er å sette navn på forskjellige begreper som brukes ved vedlikehold i et distribuert system.
- Kapittel 7 tar for seg ulike utfordringer ved det å bruke mobile agenter til vedlikehold. I dette kapitlet foreslår jeg også en måte å kategorisere endringer som kan utføres av mobile agenter.
- Kapittel 8 tar for seg oppsummering og konklusjon av oppgaven. Kapitlet inneholder også forslag til videre arbeid innen dette feltet.
- Vedlegg A beskriver forskjellige tekniske sider ved MORP. Dette gjelder blant annet hvordan MORP kan startes. Komplett kildekode finnes også i dette vedlegget.
- Vedlegg B inneholder klassene som utgjør grensesnittene mellom klient og tjener i MORP.

Kapittel 2

Evolusjon og vedlikehold av programvare

There is no such thing as a 'finished' computer program.
[TG96, side 1]

Få eller ingen systemer over en viss størrelse er ferdig den dagen de blir tatt i bruk. Dette understrekes av sitatet over som er hentet fra en undersøkelse rundt utvikling og vedlikehold av systemer [LB85]. Etter at et system er tatt i bruk oppstår det ofte et behov for å rette feil, tilpasse systemet eller legge til nye funksjoner. Dette kalles vedlikehold.

I dette kapitlet ser jeg nærmere på hva som menes med utvikling og vedlikehold av programvare. Kapitlet er organisert i to deler:

- Første del er en generell diskusjon om dataprogrammers natur, og om hvordan systemer utvikler seg gradvis. Mer konkret vil jeg se på forskjellige modeller for systemutvikling. Disse modellene sier noe om hvordan vi bør organisere arbeidet med å utvikle programmer.

- Andre del ser på hva vedlikehold er, hvilke typer vedlikehold vi har og hva som er hovedproblemene med vedlikehold. Kapitlet oppsummerer med å se på forskjellige teknikker for å bedre vedlikeholdbarheten i et program.

Teoriene som presenteres i dette kapitlet brukes videre i kapittel 6 til å presentere en strategi for hvordan mobile agenter kan brukes til å utføre vedlikehold.

2.1 Programevolusjon

Det å utvikle programvare har mange likhetstrekk med andre ingeniørfag som for eksempel det å utvikle en bil eller bygge et hus. Felles for disse er at de ofte begynner med en spesifikasjon som forteller utviklerne om ønskelige krav til systemet, bilen eller huset.

Etter spesifikasjonsfasen er det vanlig å finne ut av hvordan kravene kan oppfylles. Dette kalles *design*. Her finner man for eksempel ut hvilken motor bilen skal ha eller hvordan bilen skal se ut.

Biler og hus er kompliserte konstruksjoner, og de krever god planlegging før man begynner å utvikle dem. I tillegg har datasystemer ekstra egenskaper som gjør dem vanskelige å utvikle. I artiklen *No Silver Bullet: Essence and Accidents of Software Engineering* av Frederick P. Brooks Jr. [Jr.87] trekkes fire av disse egenskapene fram:

1. Programmene vi lager er komplekse sammenlignet med andre konstruksjoner som biler og bygninger. Få deler i koden er like, og vi kan ikke uten videre bruke kode fra andre programmer. Her er det stor forskjell sammenlignet med biler og hus hvor ferdige komponenter kan brukes i langt større grad.

   Et program kan innta mange forskjellige tilstander. Dette gjør at det er vanskelig å forstå, beskrive og teste programmet. Videre er det ofte slik at forskjellige deler av programmet samvirker med hverandre på forskjellige måter. Ved å legge til nye moduler vil kompleksiteten stige enda mer.

   Med kompleksiteten kommer problemet med å organisere et prosjekt og medlemmene i det. Store programmer fører til at kommunikasjonen mellom de forskjellige prosjektmedlemmene blir vanskelig. Utskiftninger av personer fører også til at nye personer på prosjektet må lære seg systemet fra bunnen av.

2. Programmene vi lager må være i samsvar med den virkelige verden. Følgelig må programvaren tilpasses andre systemer og ikke nødvendigvis bare datasystemer. Når bruksområdet for et program endres, må også programmet forandres.

3. Programmer er ofte utsatt for endringer. Dette kan være feilretting eller utvidelser. Hvis man sammenligner programmer med biler, er det sjelden at biler endres etter at de er levert fra fabrikken. Dette skjer derimot ofte med programvare. Grunnen til dette kan være at programkode er lettere å endre på enn en bil.

   Artiklen [Jr.87] hevder også at vellykkede programmer må bli endret. Vellykkede programmer vil si programmer som gjør det de skal og

som blir tatt i bruk. Når brukere har funnet ut at de liker et program, vil de ofte prøve å finne nye bruksområder for det. Programmet må da endres.

4. Programmer er vanskelige å visualisere. I motsetning til et hus eller en bil, må programmer visualiseres på en annen måte. Dette fører til at hjernen ikke klarer å ta i bruk de konseptuelle egenskapene på samme måte som når man tegner en bil. Dessuten fører det til hindring i kommunikasjon mellom utviklere.

Programvare er av en slik natur at det er vanskelig å kommunisere mellom utvikler og sluttbruker hva som er de egentlige kravene til systemet. Man må ha et språk eller en metode for å beskrive hva programmet skal gjøre. Dette er spesielt viktig for å finne fram til hva slags oppgaver programvaren skal løse. Forskjellige brukergrupper kan ha motstridende meninger om hva systemet skal gjøre. Dette kan føre til uklare spesifikasjoner for et program. I tillegg kan forhåpningene til et program være såpass store at kundene forventer et program perfekt tilpasset bruksområdet.

Vi kan si at programvare eller systemer utvikler seg via evolusjon. Dette vil si at et system gradvis blir forandret. Som en konsekvens av dette vil kompleksiteten på systemet øke. Neste avsnitt tar for seg hva som skjer når store systemer endres.

2.1.1 Lehmans lover om programevolusjon

Lehman og Belady [LB85] har foretatt undersøkelser av en rekke store systemer. Ut fra disse undersøkelsene har forfatterne satt opp en del hypoteser som blir kalt for *Lehmans lover* [Som96, side 664]. Dette er observasjoner som har vist seg å stemme bra for en rekke store systemer i endring.

Den første observasjonen til Lehman og Belady er at et system er i kontinuerlig forandring. Dette forklares ved at et program som brukes daglig må endres fordi miljøet rundt systemet forandres. Systemet må tilpasses miljøet ellers bli mer og mer unyttig. I [DM93, side 77] trekkes det fram at måten vi tradisjonelt bygger systemer på er basert på oppdeling i moduler og strukturering. Som en konsekvens av dette passer ikke tradisjonelle systemer spesielt bra til mer organiske organisasjoner som er avhengig av å være mer fleksible med hensyn til endringer.

Den andre observasjonen i *Lehmans lover* sier at strukturen i et system etter hvert vil bli ødelagt på grunn av endringer. Med strukturen i et system menes for eksempel klassehierarkiet eller samspillet mellom funksjoner. Strukturen blir ødelagt fordi systemet blir stort og uoversiktlig. For å ta vare på strukturen må det settes av ekstra ressurser til å reorganisere koden

slik at den blir mer oversiktlig.

Den tredje observasjonen sier at programvare over en viss størrelse har en egen dynamikk. Denne dynamikken blir bestemt tidlig i utviklingen av systemet, og den legger begrensninger på vedlikehold og mulige endringer. Dette kan begrunnes med at store systemer er såpass uoversiktlige at innføring av en funksjon kan introdusere feil andre steder i systemet. Størrelsen på systemet kan altså være et hinder for videre utvidelse. Lehman og Belady foreslår at denne tredje observasjonen kan skyldes begrensninger i strukturen på systemet [Som96, side 665].

Den fjerde observasjonen sier at på lang sikt har ressursene satt av til utvikling lite å si for hvordan systemet utvikler seg. Dette kan begrunnes med at man tidlig i et prosjekt setter rammene for hvordan systemet kan utvikle seg. Dette stemmer med den tredje observasjonen. Det blir også trukket fram at store prosjekter med mange ansatte vil være ulønnsomme på grunn av det store kommunikasjonsbehovet mellom ansatte. Et utviklingsprosjekt når en grenseverdi hvor det ikke vil være lønnsomt å ansette flere personer.

Den femte og siste observasjonen til Lehman og Belady sier at for hver ny versjon av et system, vil omfanget av endringene være konstant. I følge den tredje observasjonen kan innføringen av en ny funksjon føre til feil i andre deler av systemet. Dette vil si at en ny versjon med nye funksjoner også vil føre til en del feil. For å rette på disse feilene er det nødvendig å gi ut enda en ny versjon av systemet. Sommerville peker derfor på at det er en dårlig strategi å blande feilretting og innføring av nye funksjoner i en og samme versjon [Som96, side 687].

2.1.2 Systemutviklingsmodeller

For å organisere utviklingsprosessen har man kommet fram til ulike modeller for å utvikle systemer. Disse kalles for *systemutviklingsmodeller*.

En systemutviklingsmodell systematiserer måten man utvikler programmer på. Generelt består disse modellene av et antall aktiviteter eller faser. Disse aktivitetene er forskjellige fra modell til modell, men de har også likhetstrekk. De vanligste aktivitetene er oppsummert i tabell 2.1.

I sin artikkel om spiralmodellen presenterer Boehm [Boe86] fire forskjellige utviklingsmodeller. Disse modellene går under navnene *code-and-fix*, fossefallsmodellen, prototyping[1] og spiralmodellen.

Code-and-fix modellen går ut på å skrive en del kode, kompilerer og teste om programmet virker. Alle har en eller annen gang jobbet på denne måten, selv om dette ofte er en ineffektiv måte å jobbe på. Programmet kan etter

[1] Prototyping blir kalt evolusjonær modell i [Boe86].

Aktivitet	**Forklaring**
Analyse	Analysefasen går ut på å finne fram til de egentlige kravene i et system. Det er ikke alltid kunden eller oppdragsgiver vet hva de vil at systemet skal gjøre. Kanskje er det andre forhold i organisasjonen eller bedriften som må endres?
Spesifikasjon	Etter analysefasen er man nødt til å finne fram til *hva* systemet skal gjøre.
Design	Designfasen går ut på å finne fram til *hvordan* man skal utvikle systemet. Dette innebærer å finne fram til en fornuftig oppdeling av systemet i moduler. Grensesnittene mellom moduler samt grensesnittet mot eksterne systemer må også kartlegges.
Implementasjon	Systemet må programmeres i henhold til spesifikasjonene og designet.
Testing	For å luke bort mest mulig feil utføres ulike tester av systemet. Det vanlige er å teste små moduler, sette dem sammen, teste disse og så videre. Ytelse og hvor mye belastning systemet tåler må også testes.
Integrering	Man må ofte integrere systemet med andre eksisterende systemer.
Installering	Dette innebærer ikke bare installering av systemet på kundens maskinvare, men også opplæring i hvordan systemet skal brukes.
Systemevolusjon	Etter at programmet er tatt i bruk kan det være behov for å rette feil eller å legge til nye funksjoner. Dette kalles også programvarevedlikehold eller bare vedlikehold.
Utfasing av systemet	På et tidspunkt har systemet utspilt sin rolle og andre nye systemer tar over dets funksjon. Ved utfasing kan man komme ut for problemer som datakonvertering og parallell drift av to systemer.

Tabell 2.1: Tabellen viser en oversikt over mulige aktiviteter i en systemutviklingsmodell.

hvert bli rotete og vanskelig å forstå. Etter at en del av koden virker, bygger man videre på denne. Dette fører ofte til en usammenhengende struktur som er vanskelig å vedlikeholde. Denne modellen inneholder heller ingen fase for å finne fram til kravene til programmet. Det kan derfor hende at det ferdige programmet ikke svarer til kravene brukerne har.

Fossefallsmodellen inneholder forskjellige faser satt opp i en rekkefølge. Fasene i modellen kan deles inn i analyse, design, implementasjon og testing. Modellen inneholder også mulighet for å gå tilbake ett eller flere steg. Det gjør at utvikling kan skje iterativt eller i flere omganger. Problemet med denne modellen er at den baserer seg på at all dokumentasjon for en aktivitet må være ferdig før neste aktivitet kan starte. Hvis dokumentasjonen til en av de tidligere aktivitetene bygger på gale forutsetninger, kan man risikere å ende opp med et program som ikke tilfredsstiller kravene fra brukerne. Den sekvensielle utføringen fører også til problemer hvis det er nødvendig å gå tilbake ett eller flere steg i modellen.

Prototyping går ut på at man gjennom hele utviklingsprosessen har en prototype til demonstrasjon for brukerne. Dette fører til at utviklingstemaet får verdifulle erfaringer om hvordan programmet bør være, men denne modellen har de samme problemene som *code-and-fix* modellen. Begrunnelsen for dette er at når en av prototypene virker, er det fristende å bruke denne som grunnlag for videre utvikling. Problemet med dette er at prototypen er programmert spesielt for demonstrasjoner. Det kan derfor hende at utviklerne har brukt en del snarveier under design og implementasjon for å få programmet til å virke. Det er ikke sikkert at disse løsningene egner seg i det endelige programmet. En måte å unngå at prototypen brukes i det ferdige produktet, er å programmere prototypen i et annet programmeringsspråk enn det endelige programmet.

Spiralmodellen er en iterativ modell med fokus på å identifisere mulige risikoer. Hver iterasjon i modellen består av fire faser:

1. fastsett mål, begrensninger og alternativer for denne iterasjonen,
2. evaluer alternativer, identifiser risikoer og mulige løsninger på dem,
3. utvikle og verifisere programmet,
4. planlegg neste fase.

Fordelen med modellen er at den er fleksibel og kombinerer mange av fordelene fra de andre modellene. En ulempe ved spiralmodellen er at man må finne fram til de største risikoene og løse disse først. Som regel er det fristende å løse de minste problemene først og vente med å løse de vanskelige [TG96, side 37].

I motsetning til fossefallsmodellen baserer ikke spiralmodellen seg på at dokumentasjonen for hvert steg må være ferdig før man går videre til neste steg. Selv om det er mulig å bruke fossefallsmodellen iterativt, er den uegnet til prosjekter hvor spesifikasjonene forandres underveis.

Systemutviklingsmodellene i dette kapitlet er modeller av hvordan arbeidet med å utvikle et program kan deles inn. Det er også viktig å få fram at det ikke er nødvendig å følge aktivitetene i tabell 2.1 slavisk. Etter min erfaring tilpasses modellene etter størrelse og type prosjekt. I prosjekter jeg har vært med på har vi startet med en modell som ligner prototyping. Begrunnelsen for dette kan være at kravene til programmet er uklare. En prototype kan være med på å klargjøre målene med programmet. Etter at kravene er fastlagt, kan man bruke en systemutviklingsmodell som ligner mer på spiralmodellen.

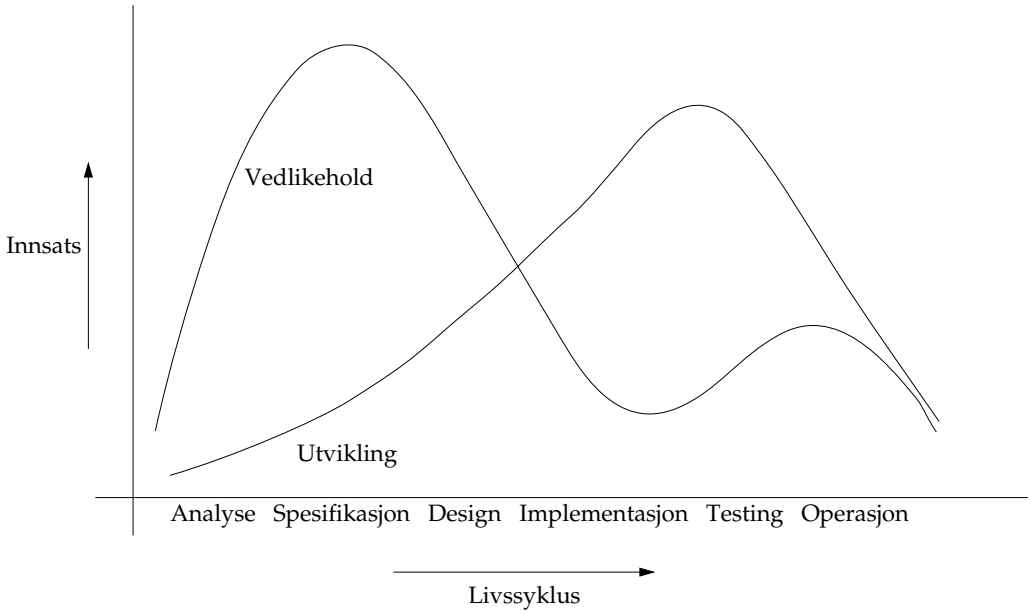
Spiralmodellen fremhever en *iterativ* og *inkrementell* arbeidsmåte. Generelt går en iterativ arbeidsmåte ut på å gå gjennom alle aktivitetene i tabell 2.1 flere ganger. I løpet av de første iterasjonene vil fokus ligge på analyse og spesifikasjon. For hver iterasjon må man innom hver aktivitet. Fordelen ved å utføre utviklingen iterativt er at det er mulig å fange opp manglende krav til systemet. Modellen er inkrementell fordi en iterasjon bygger på resultatene fra den foregående. Nyere prosesser for systemutvikling benytter dette. Et eksempel er *Rational Unified Process* [BRJ99, side 449].

2.2 Programvedlikehold

Vedlikehold kan ikke sammenlignes med det å utvikle et system for første gang. Det er fordi det er vanskeligere å legge til ny funksjonalitet etter at systemet er tatt i bruk. Begrunnelsen for dette kan være at systemet er dårlig dokumentert eller at omfattende tester må gjennomføres. I [TG96, side 41] blir det trukket fram at tradisjonelle modeller ikke tar hensyn til at systemer skal videreutvikles. Det er derfor et behov for systemutviklingsmodeller som fanger opp den evolusjonære naturen til et system.

Figur 2.1 fra [TG96, side 41] viser at det er viktig å fokusere på vedlikehold allerede i de første fasene av en utviklingsmodell. Grunnlaget for vedlikeholdbarheten til programmet legges i løpet av disse fasene. Analysefasen er viktig fordi man må finne fram til hva som er den egentlige hensikten med systemet. Dette er ofte et problem fordi de som bestiller systemet ikke helt vet hva systemet skal brukes til [PC86]. Dette kan skyldes at denne typen system ikke har vært utviklet før.

Designfasen er viktig fordi det er i løpet av denne fasen strukturen og arkitekturen til programmet blir bestemt. En lite gjennomtenkt arkitektur kan legge begrensninger på hvordan det er mulig å endre programmet.



Figur 2.1: Figuren viser hvilke faser som bør vektlegges for å bedre vedlikeholdbarheten i et system [TG96, side 41].

Sommerville [Som96, side 660] forklarer programvedlikehold som prosessen det er å endre et system etter at det er levert til kunden. Endringene kan være av forskjellig art, men de har som mål å bedre kvaliteten på systemet. En mer presis definisjon av programvedlikehold finnes i [TG96, side 3]:

> *[Software maintenance is defined as:] modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.*

Det er mange som ser på programvare som kun kildekoden. Dette er feil i følge [TG96, side 2]. Programvare bør omfatte all dokumentasjon av systemet samt instruksjoner om hvordan systemet skal brukes. Når koden forandres bør også den tilhørende dokumentasjonen oppdateres. Dette er et viktig poeng siden foreldet og manglende dokumentasjon er et av hovedproblemene i programvedlikehold [TG96, side 22].

Årsakene til at vedlikehold utføres er mange. Disse kan grovt deles opp i fire kategorier [TG96, side 4]:

- Ofte er det slik at store systemer som brukes i industrien eller i samfunnet er kritisk for virksomheten. En del vedlikehold går derfor ut på å rette feil slik at tjenestetilbudet blir opprettholdt. Dette kalles for *korrektiv endring*. I denne kategorien inngår enkle kodefeil, designfeil

og logiske feil.

- En annen type vedlikehold utføres når man oppgraderer systemet slik at det er tilpasset nye lover eller bestemmelser. Innføringen av den felles europeisk valutaen, euro, er et eksempel på en slik oppgradering. Denne kategorien av vedlikehold kalles *adaptiv endring*.
- Ofte er det slik at systemer utvides selv om de fungerer bra. Dette kan være for å bedre ytelsen, tilpasse systemet eller legge til ny funksjonalitet. Denne typen endring kalles *perfektiv endring*.
- Man kan utføre vedlikehold for å bedre strukturen på et system slik at fremtidig vedlikehold blir lettere. I følge den andre observasjonen i *Lehmans lover* vil stadige utvidelser føre til en ødelagt struktur (se side 14). Det må derfor ekstra innsats til for å bedre strukturen. Dette typen endring betegnes som *preventiv endring*.

Selv om de forskjellige vedlikeholdskategoriene ofte går over i hverandre, er det viktig å skille mellom dem. For eksempel er korrektive endringer viktigere enn det å legge til en ny funksjon. Dette gjør det lettere å prioritere viktige endringer framfor mindre viktige [TG96, side 9].

Ved å kategorisere de forskjellige endringene kan man også finne ut hvilken type endring som utføres mest. Undersøkelser har vist at perfektive endringer er den vedlikeholdstypen som blir utført mest (opptil 50 prosent) [TG96, side 13] og [Som96, side 660].

Dette betyr at en stor del av vedlikeholdsarbeidet går ut på å legge til nye funksjoner i et system. Store perfektive endringer kan betraktes som selvstendige utviklingsprosjekter. Grunnen til dette er at utviklerne må gjennomføre en eller flere av aktivitetene nevnt i tabell 2.1. I tillegg er det nødvendig å analysere hvordan utvidelsene vil påvirke andre deler av systemet. Som oftest er det umulig å finne fram til alle konsekvensene. Det er derfor nødvendig å teste systemet etter at nye funksjoner er lagt til.

I en artikkel om vedlikehold og konkurransedyktighet [O'N97] hevdes det at det er en fordel om bedrifter har systemer som er lette å endre. Grunnen til dette er at bedrifter raskt vil tilpasse seg nye lover og bestemmelser for ikke å miste gevinst eller markedsgrunnlag. Vedlikeholdbare systemer kan ses på som en konkurransefordel.

2.2.1 Hva er problematisk med vedlikehold?

Å endre et system kan ta lang tid, og det kan være kostnadskrevende. Det er flere faktorer som påvirker vedlikehold og kostnadene ved vedlikehold. Disse faktorene kan grovt deles inn i tekniske og ikke-tekniske faktorer [Som96]. Disse faktorene er gjengitt i tabell 2.2.

Alle systemer som blir endret bør testes. Hvor grundig testingen er, avhenger av type system. Det kan derfor være dyrt å legge til nye funksjoner etter at testingen er avsluttet.

Selv om noe vedlikehold i seg selv kan betraktes som et fullverdig prosjekt, viser det seg at langt mindre ressurser blir satt av til dette arbeidet i forhold til ved utviklingen av systemet. Ofte er det slik at vedlikehold blir sett på som annenrangs arbeid i forhold til nyutvikling. Derfor settes uerfarne utviklere til å gjøre vedlikehold. Vedlikehold har derfor et dårlig rykte [Som96, side 660].

2.2.2 Hvordan bedre vedlikeholdbarheten?

Det finnes flere teknikker for å gjøre et system lettere å vedlikeholde. En viktig faktor, som ikke vil bli tatt opp så nøye i denne oppgaven, er kvalitetskontroll. Dette innebærer kontroll av programvare samt selve prosessen som brukes til å utvikle programmene. Omfanget av kvalitetskontroll avhenger av størrelsen på organisasjonen og typen programvare de utvikler.

En mer konkret teknikk for å bedre vedlikeholdbarhet baserer seg på gjenbruk. I [RWL95, side 135] blir det skilt mellom *tilfeldig* og *planlagt* gjenbruk. Et eksempel på tilfeldig gjenbruk er hvis man bruker kode fra et annet program. Planlagt gjenbruk går ut på å utvikle generelle komponenter eller kodebiblioteker som igjen kan brukes i forskjellige systemer.

Idéen med gjenbruk kan utnyttes på forskjellige nivåer i et utviklingsprosjekt [RWL95, side 136]:

- Ved analyse av et domene kan man bruke lignende analyser fra andre systemer. For eksempel er det mulig å konstruere et kontosystem for banker utfra en mer generell modell for kontosystemer. En samling av objektmodeller for forskjellige domener finnes for eksempel i [Fow96] og [CLL99]. Disse gjenbrukbare objektmodellene kalles ofte analysemønstre[2].

- En annen form for gjenbruk er designmønstre[3]. Et designmønster beskriver et generelt problem, løsningen på problemet og i hvilken kontekst løsningen fungerer [Joh97]. En samling designmønstre finnes for eksempel i [GHJV96].

- Deler av et system kan implementeres ved at man bruker *komponenter*. En gjenbrukbar komponent er et produkt som løser en spesiell

[2] Analysemønstre er min oversettelse av *analysis patterns*.
[3] Designmønstre er min oversettelse av *design patterns*.

Tekniske faktorer	
Modul uavhengighet	Hvor avhengig er modulene av hverandre? Det skal være mulig å forandre en modul uten at dette påvirker hele systemet. Man skiller gjerne mellom grensesnittet og implementasjonen av grensesnittet til en modul. All kontakt med resten av systemet bør gå gjennom grensesnittet.
Programmeringsspråk	Hvilket språk er systemet programmert i? Man skiller grovt mellom høynivå og lavnivå programmeringsspråk.
Kodestandard og form	Valg av programmeringsform har mye å si for forståelsen av programmet. En kodestandard angir for eksempel hvordan variable skal navngis eller hvordan koden skal kommenteres.
Validering og testing av programvare	Et program som er testet grundig har færre feil enn et program som ikke er testet. Ved grundig testing er det mulig å redusere utgiftene til korrektive endringer.
Kvalitet på dokumentasjon	For å utføre vedlikehold på et system er det nødvendig å forstå systemet. En god dokumentasjon gjør dette lettere.
Konfigurasjonsstyring og versjonskontroll	Både systemet og dokumentasjonen utvikler seg i mange forskjellige versjoner. For å holde orden på dette er det viktig med et verktøy for versjonskontroll. Konfigurasjonsstyring handler om å administrere forskjellige sammensetninger eller utgaver av et system (for eksempel for ulike operativsystemer).

Ikke-tekniske faktorer	
Applikasjonsdomenet	Noen applikasjoner har blitt utviklet såpass ofte at det er liten tvil om hva systemet skal gjøre (for eksempel et tekstbehandlingssystem). Innen andre domener finnes det få eller ingen erfaringer.
Ansatte	Det er enklere å vedlikeholde et program man har skrevet selv. Det tar blant annet kortere tid å forstå programmet.
Alderen på programmet	Strukturen på et program kan være ødelagt hvis det er utført mye vedlikehold.
Avhengigheter til eksterne systemer	Det kan hende at endringer i et eksternt system fører til endring i det aktuelle systemet.
Maskinvare	Programmer må ofte endres for å ta i bruk ny maskinvare.

Tabell 2.2: Forskjellige faktorer som påvirker vedlikehold av et system [Som96, side 667].

klasse av problemer for et bestemt formål [RWL95, side 135]. En komponent kan for eksempel være koden for en kalender. Et eksempel på en modell for komponenter er JavaBeans [Fla97, side 178].

Samlinger av gjenbrukbare mønstre på forskjellige nivåer kalles for *rammeverk* [Joh97]. Et rammeverk for et lagersystem vil for eksempel inneholde et analysemønster som er et forslag til objektmodell for systemet. Objektmodellen må tilpasses slik at den passer til systemet som skal konstrueres. Videre kan et rammeverk inneholde kode i form av et klassebibliotek. Dette klassebiblioteket kan brukes direkte eller tilpasses ved at man spesialiserer klasser.

Forskjellige typer gjenbrukbare mønstre presenterer løsninger på generelle problemer. Disse generelle mønstrene må tilpasses hvert enkelt system.

Samlinger av analysemønstre og designmønstre er verdifulle fordi de representerer erfaringer andre har gjort med lignende problemer. I [Som96, side 397] nevnes flere fordeler med gjenbruk og da kanskje spesielt gjenbruk av kode:

- Påliteligheten til systemet kan bli bedre fordi ferdige komponenter bør være bedre testet enn nye komponenter man utvikler selv.
- Risikoene ved et prosjekt kan reduseres fordi det er lettere å estimere prisen på en ferdig komponent enn prisen ved å utvikle komponenten selv.
- Spesialister kan frigjøres til andre formål enn å løse de samme oppgavene på forskjellige prosjekter.
- Forskjellige standarder kan brukes i komponentene. Dette gjelder for eksempel standarder for hvordan brukergrensesnittet skal se ut.
- Utviklingstiden på et prosjekt bør kunne reduseres ved å ta i bruk ferdige komponenter.

Gjenbruk av analyse, kode og design er en fordel fordi vi kan basere oss på vellykkede erfaringer fra andre. Dette kan føre til at vi slipper å gjøre de samme feilene som andre har gjort.

2.3 Oppsummering

Dette kapitlet har sett på hvordan systemer utvikler seg. Observasjonene i [LB85] og [Jr.87] har vist at store systemer har sin egen dynamikk som i

stor grad bestemmer hvordan og hvor mye et system kan endres. I store systemer kan endringer føre til uventede feil andre steder i systemet.

Forskjellige systemutviklingsmodeller forsøker å strukturere måten vi utvikler programvare på. De fleste modellene er inndelt i aktiviteter som setter fokus på analyse, kravspesifikasjon, design, implementasjon og testing. De fleste modellene legger vekt på å gi en grundig forståelse av problemområdet.

Vedlikehold av programvare skiller seg fra vanlig utvikling fordi man endrer systemet etter at det er levert til kunden. Dette stiller spesielle krav til de som skal utføre vedlikeholdet. Videre deles vedlikehold inn i fire hovedkategorier. Den dominerende av disse er perfektiv endring som går ut på å utvide eller forbedre funksjonaliteten i et system.

Teknikker for å bedre vedlikeholdet tar ofte utgangspunkt i forskjellige former for gjenbruk. Dette kan være med på å bedre vedlikeholdbarheten til en applikasjon.

Kapittel 3

Mobile agenter

In contrast to the software objects of object-oriented programming, agents are active entities that works according to the so-called Hollywood principle: "Don't call us, we'll call you!"

[LO98, side 2]

Begrepet mobil agent og agent brukes i mange sammenhenger både i reklame for programvare og i forskning. Den overbelastede bruken av begrepene har ført til at det er vanskelig å vite hva som egentlig menes med en agent eller en mobil agent. Dette kapitlet vil derfor begynne med å se på bakgrunnen for agenter.

Videre vil kapitlet forklare hva som menes med en mobil agent. Mobile agenter vil bli definert ut fra en konseptuell og en teknisk synsvinkel. Egenskapene til mobile agenter blir også diskutert.

For at mobile agenter skal kunne migrere mellom forskjellige maskiner er det nødvendig med programvare som støtter dette. Denne programvaren kalles for et *agentsystem*. Agentsystemet tilbyr agentene en kjøreomgivelse samt forskjellige tjenester. Samtidig skjuler agentsystemet en del av detaljene i operativsystemet og nettverket slik at det blir lettere å programmere mobile agenter.

Siste del av kapitlet vurderer mobile agenter opp mot mer tradisjonelle distribuerte systemer. Emner som diskuteres her er egenskaper ved distribuerte systemer og ulike kommunikasjonsmodeller.

3.1 Kort om bakgrunnen for mobile agenter

Begrepet agent stammer fra forskning innen distribuert kunstig intelligens. Her utviklet man en utvidet objektmodell som ble kalt *concurrent actor model* [Nwa96]. Denne modellen innførte begrepet *aktører*. Aktører har i likhet

med objekter intern tilstandsinformasjon og innkapsling av data, men i tillegg kan aktører svare på meldinger fra andre aktører. Aktørene utfører også sitt arbeid i parallell.

Det er stor likhet mellom aktørbegrepet og det vi i dag kaller en agent. Dette innebærer i følge [WJ95] at agenten er:

Autonom: eller selvstendig. Dette betyr at agenten opererer uten interaksjon fra mennesker, og agenten har kontroll over intern tilstand samt egne handlinger.

Sosial: det vil si at agenten kan samarbeide med andre agenter ved hjelp av et kommunikasjonsspråk for agenter. I dette perspektivet kan en agent også være et menneske. Kommunikasjonen kan da skje via et brukergrensesnitt.

Reaktiv: agenten oppfatter hendelser i sine omgivelser og handler på bakgrunn av disse.

Pro-aktiv: dette vil si at agenten ikke bare reagerer på hendelser i omgivelsene, men selv tar initiativet til å løse sine oppgaver. Agenten har et mål.

En agent med egenskapene som beskrevet over blir betegnet som *weak agency* [WJ95]. *Strong agency* er agenter som i tillegg til egenskapene over har egenskaper vi vanligvis forbinder med mennesker. Dette kan for eksempel være kunnskap, tro, intensjon og plikter.

Forskningen på agenter kan grovt deles i to grupper [TT97]:

1. intelligente agenter, og
2. mobile agenter.

Forskningen på intelligente agenter fokuserer på hvordan agenter kan anvendes til brukerstøtte og informasjonsbehandling. Spesielt ser man på hvordan grupper av agenter kan samarbeide om å løse oppgaver. Her er det et klart skille mellom å se på en gruppe agenter i motsetning til en enkelt agent [Nwa96]. Et eksempel på et bruksområde for en gruppe av agenter er informasjonsinnsamling og filtrering. Dette kan for eksempel brukes ved søk i store informasjonsmengder. Et eksempel på bruksområde for en enkelt agent er filtrering av e-post for en bruker.

Mobile agenter har ikke utgangspunkt i forskningen rundt kunstig intelligens, men i distribuerte systemer [WPM99]. De har utviklet seg fra forskningen rundt prosessmigrering, fjernevaluering og mobile objekter. Målet med denne forskningen har vært å finne fram til mer fleksible løsninger for

kommunikasjon enn den tradisjonelle klient/tjener modellen [MGW97]. Et eksempel på dette kan være det å flytte en prosess fra en belastet maskin over til en annen maskin med mindre belastning.

Begrepet intelligente agenter blir ofte brukt i forbindelse med mobile agenter. Jeg mener at intelligens i denne sammenheng ikke kan sammenlignes med den intelligens mennesker har. I innledningen til et spesialnummer om agenter (*IEEE Internet Computing*, juli-august 1997) blir intelligens diskutert av C.J. Petrie [Int97]. Konklusjonen er at vi ønsker smartere programmer som kan hjelpe til med å organisere komplekse systemer og store mengder data. Smartere programmer kan være programmer som koordineres i forhold til hverandre, tar egne avgjørelser eller forflytter seg for å redusere nettverksforsinkelsen.

Mobile agenter trenger ikke å være intelligente, og de færreste mobile agenter oppfyller kravet til definisjonen som brukes på agenter innen kunstig intelligens [TT97]. Mobile agenter ligger innenfor kravene til *weak agency*.

3.2 Hva er en mobil agent?

Det er ikke lett å svare på spørsmålet "Hva er en mobil agent?". For det første finnes det mange definisjoner som legger vekt på de forskjellige egenskapene en mobil agent bør ha. Videre er det viktig å forklare hva som skiller en mobil agent fra et hvilket som helst annet program. Hvorfor kalles det en mobil agent og ikke et program?

Forklaringen ligger delvis i at det finnes flere perspektiver på hva en mobil agent er. En sluttbruker kan ha dette perspektivet på hva en agent er [Lan98]:

> *An agent is a program that assists people and acts on their behalf. Agents function by allowing people to delegate work to them.*

Sentralt i forklaringen er at en agent er til for å hjelpe oss som brukere. Agenten må ha tillatelse til å utføre oppgaver på vegne av en bruker. Denne forklaringene legger vekt på hva en bruker forventer av funksjonalitet, men den sier ikke hvilke egenskaper en agent bør ha. Det er også viktig å legge merke til at dette er en generell forklaring på hva en agent er, og da ikke nødvendigvis agenter som brukes i databehandling.

Egenskapene til en mobil agent kan beskrives ut fra det som blir kalt systemperspektivet [Lan98]. Disse egenskapene er de samme som beskrevet for *weak agency* på side 26. I tillegg må en mobil agent ha mulighet til å forflytte seg mellom forskjellige maskiner eller prosesser.

Fra et teknisk perspektiv kan en mobil agent defineres på denne måten [Lan98]:

> *A mobile agent is not bound to the system where it begins execution. It has the uniqe ability to transport itself from one system in a network to another. The ability to travel, allows a mobile agent to move to a system that contains an object with which the agent wants to interact, and then to take advantage of being in the same host or networks as the object.*

Denne definisjonen legger vekt på at den mobile agenten skal være uavhengig av prosessen som instansierte agenten. Vi sier at en mobil agent *migrerer* når den flyttes fra en prosess til en annen.

At agenten migrerer vil si at den skifter adresserom. Når en mobil agent skal eksekvere trenger den sin egen kode, sin tilstand og plass til data som agenten genererer [FPV98]. Ved migrering må denne informasjonen flyttes til en annen prosess.

Et problem ved migrering er å opprettholde de bindinger agenten har mot prosessen den kom fra. Dette kan være kommunikasjon mot andre prosesser eller referanser til lokale objekter. Migrering handler derfor også om å gjøre om lokale objektreferanser til fjernreferanser slik at agenten kan opprettholde sine bindinger.

Definisjonen over sier også at agenten selv skal kunne transportere seg selv fra maskin til maskin. Dette innebærer at agenten inneholder regler for når den skal forflytte seg selv. Videre må agenten vite hvilken maskin den skal til neste gang. Agenten kan være forhåndsprogrammert med adresser eller den kan få sin neste adresse ved hjelp av forskjellige navnetjenester.

En mobil agent må opptre på vegne av andre, den må være selvstendig (autonom), den må ha en oppgave (pro-aktiv), den må reagere på hendelser i omgivelsene (reaktiv) og den må ha mulighet til å migrere. I tillegg kan en mobil agent samarbeid med andre agenter samt ha evnen til å lære.

3.2.1 Hva skiller en mobil agent fra et program?

En mobil agent består i likhet med et program av kode, datalager og kjøretilstand. Det som skiller en mobil agent fra et program er at en agent er skrevet spesielt for å kunne flyttes rundt i et nettverk. For at dette skal være effektivt må en agent være mindre enn et vanlig program. Som oftest utgjør en agent en liten del av et større program [TT97]. Det å flytte en agent rundt til forskjellige maskiner er ikke prinsipielt forskjellig fra det å flytte en fil ved hjelp av filoverføring. I likhet med filoverføring blir agenter også overført via tradisjonelle kommunikasjonsprotokoller som for eksem-

pel TCP/IP[1]. I tillegg må mobile agenter fryses eller serialiseres. Dette vil si at koden og variablene til agenten blir gjort om til et standard format slik at agenten kan skrives til fil eller sendes over et nettverk til en annen prosess.

Når det gjelder tilstanden til en agent skiller man mellom to typer agenter [FPV98]:

1. sterke agenter, og
2. svake agenter.

En sterk agent tillater at kode og kjøretilstand kan flyttes til en annen prosess. Svake agenter tillater kun at kode og noe data kan flyttes. Dette begrenses i stor grad av hvilket programmeringsspråk som brukes og hvordan kjøretidssystemet er.

Rent teknisk skiller ikke mobile agenter seg fra andre programmer, men mobile agenter er en ny måte å organisere kommunikasjonen på i forhold til klient/tjener modellen. Mer om dette på side 42.

3.3 Bruksområder

Mobile agenter har ofte blitt beskyldt for å være "løsningen uten et problem" [MGW97]. Påstanden er at det finnes andre teknikker som løser de samme problemene som mobile agenter løser. Hovedkritikken når det gjelder mobile agenter er at det vil være upraktisk å flytte kode og tilstand over til en annen maskin. Dette er upraktisk fordi man bruker lengre tid og mer båndbredde på migreringen enn ved å bruke en alternativ teknikk. Et alternativ kan for eksempel være å bruke vanlig meldingsutveksling eller stasjonære agenter.

Hvis man sammenligner mobile agenter med dagens protokoller for datakommunikasjon, er det klart at mobile agenter krever mer båndbredde. Derfor vil det i mange tilfeller være lønnsomt å bruke en meldingsbasert protokoll.

Denne problemstillingen blir diskutert nærmere i artiklen *Mobile Agents: Are They A Good Idea?* [CHK97]. Artiklen ser nærmere på ulike fordeler ved mobile agenter og hvordan disse kan oppnås ved å bruke andre teknikker. For eksempel trekkes det fram hvordan mobile agenter kan støtte mobile enheter som for eksempel en bærbar PC. Mobile enheter har typisk varierende tilknytning til nettverket samt begrenset kapasitet på maskinvaren.

[1] TCP står for *Transmission Control Protocol* og IP står for *Internet Protocol*. For mer informasjon om disse se for eksempel i [Tan89].

Mobile agenter kan støtte en bærbar PC ved at de kan innkapsle kommunikasjon mot tjenere i nettverket. Istedenfor å sende mange meldinger over et nettverk kan en agent brukes. Beregninger eller søk kan flyttes ved hjelp av agenten over til en tjener i nettverket. Agenten innkapsler oppgaven brukeren vil ha utført. På denne måten kan agentene redusere trafikken på nettet og avlaste den bærbare PC'en.

Som en alternativ teknikk til mobile agenter trekker artiklen [CHK97] fram proxy tjenere. Slike tjenere kan løse det samme problemet som over ved at den bærbare PC'en kun kommuniserer med proxy tjeneren. På den måten kan tjeneren samle opp informasjon og avlaste den bærbare PC'en.

Konklusjonen på artiklen er at agenter ikke er spesielt bedre egnet på individuelle områder som for eksempel det å støtte mobile enheter. Artiklen argumenterer derimot for at mobile agenter kan være et rammeverk for å løse mange problemer innen datakommunikasjon. Så istedenfor å bruke mange forskjellige teknikker kan mobile agenter brukes til å støtte for eksempel vedlikehold og mobile enheter. I prototypen i kapittel 4 er det kun sett på hvordan mobile agenter kan støtte vedlikehold.

3.3.1 Fordeler ved bruk av mobile agenter

Den fremste fordelen ved å bruke mobile agenter ser ut til å være at de kan migrere til maskinen den skal kommunisere med. Det er mye å spare i forsinkelse ved at agenten foretar en lokal kommunikasjon i forhold til å kommunisere over et nettverk, men man må også ta hensyn til den tiden og båndbredden man bruker på å frakte koden og eventuelt tilstanden til agenten over nettverket.

I [Lan98] og i [FPV98] nevnes flere fordeler ved å ta i bruk mobile agenter:

- Mobile agenter kan operere på tvers av administrative domener. Dette gjør at mobile agenter kan være velegnet for Internett.
- Distribusjonen i et program er ikke lenger transparent. Programmereren har kontroll over hvilke noder forskjellige agenter befinner seg i. Fordelen med dette er at agentene kan utnytte mulighetene som finnes på den enkelte node.
- Mobilitet kontrolleres av programmerer. Det vil si at programmereren bestemmer når agenten skal migrere. Dette kan utnyttes ved at agenten flyttes til hensiktsmessige steder (for eksempel for å korte ned forsinkelsen).
- Mobile agenter kan redusere nettverkstrafikk ved å flytte kode til stedet hvor beregningene foregår. Her er det viktig å vurdere om man

sparer mest på å la objekter ligge der de ligger eller å flytte dem over nettverket.

- Mobile agenter kan innkapsle protokoller. Applikasjoner kan oppgraderes med nye protokoller ved at agenten forflytter seg til applikasjonen.

- Mobile agenter kan bedre robusthet og feiltoleranse i programmer. Det er mulig å flytte mobile agenter over til en annen tjener eller å replikere tjenester ved å ha flere utgaver av en agent.

- Mobile agenter kan brukes til å utvikle avanserte distribuerte applikasjoner. Agenter hjelper til med å splitte opp oppgavene i mindre deler. Et komplekst problem (som for eksempel monitorering av programvare i et nettverk) kan deles opp i flere underproblemer. Agenter kan også utføre disse oppgavene i parallell.

Fuggetta *et al*. [FPV98] trekker fram sju bruksområder for mobile agenter:

Innsamling av informasjon: istedenfor å flytte store datamengder, kan man i steden flytte prosessen ut til dataene. Dette er det klassiske eksemplet som ofte brukes i forbindelse med mobile agenter. Et godt eksempel er agentsystemet TACOMA hvor agenter blant annet brukes til å beregne meterologiske data [GHN+97, side 38]. Dette er et eksempel på at beregningene (agentene) flyttes til dataene som skal beregnes.

Aktive dokumenter: typisk eksempel er en Java applet som kjører på en html side.

Avanserte telekommunikasjonstjenester: en mobil agent kan støtte brukere som er utenfor dekningsområdet til et trådløst nett. Agenten fungerer da som en proxy for meldinger til nettverket. Agent TCL er et eksempel på et agentsystem som støtter mobile enheter [KGN+97]. Andre tjenester kan være oppsett og signalering av en videokonferanse.

Kontroll og konfigurering av komponenter: dette er aktuelt i et lokalnettverk hvor man har mange forskjellige komponenter som skal styres. Kontrollprogrammer i form av agenter kan plasseres ute hos komponentene, og disse kan ha ansvaret for å rapportere om tilstanden til komponentene (for eksempel skrivere).

Samarbeid og arbeidsflyt: prosjekter består ofte av mange aktiviteter og personer. Mobile agenter kan brukes til å innkapsle aktivitetene slik at de forflyttes rundt mellom de forskjellige personene i et prosjekt.

Aktive nettverk: et eksempel på bruk er en ruter som får en pakke med en protokoll den ikke kan forstå. Ruteren kan da dynamisk laste ned protokollen den trenger. For en oversikt over aktive nettverk se [TSS+97].

Elektronisk handel: man kan gi en mobil agent rettigheter til å betale for varer eller eventuelt forhandle fram et pristilbud. Videre kan mobile agenter brukes til å flytte kontrollen nærmere informasjonen som brukes i en transaksjon. Dette er en fordel for eksempel ved handel med aksjer [WPM99].

3.4 Agentsystem

En mobil agent er avhengig av et agentsystem for å fungere. Agentsystemet tilbyr en kjøreomgivelse for agentene. Mesteparten av funksjonaliteten til en mobil agent er implementert via et agentsystem [GHN+97, side 27].

Enhver maskin som skal kunne ta i mot mobile agenter må ha et agentsystem installert. En samling agentsystemer følger ikke den tradisjonelle klient/tjener kommunikasjonen, men en *peer-to-peer* modell hvor hvert agentsystem kan spille rollen som klient eller tjener etter behov.

En generell beskrivelse av et agentsystem finnes i [KJ98, side 236]. Der presenterer man et begrepsapparat utviklet for agentsystemet *Telescript*. Lignende beskrivelser finner man i utkastet til standard for interoperabilitet mellom forskjellige agentsystemer fra OMG (Object Management Group) [MBB+98].

Foruten agenten finner man disse sentrale begrepene:

Sted: et agentsystem består av mange steder som tilbyr et oppholdssted for agenter. Man har typisk et hjemmested (hvor agenten ble sendt ut fra) og flere fremmede steder. På hvert sted kan det være agenter som igjen tilbyr tjenester.

Reise: er muligheten til å forflytte seg mellom forskjellige steder.

Møte: det er gjennom møter med andre agenter at en mobil agent kan bruke funksjonaliteten som ligger på et fremmed *sted*.

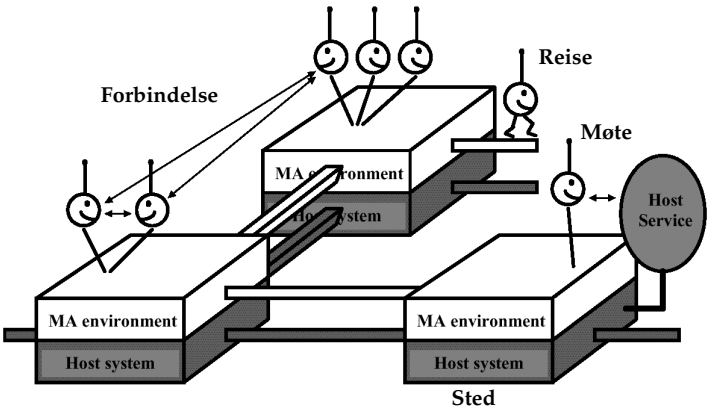
Forbindelse: en agent kan utveksle informasjon med en annen agent uten å være på samme sted.

Autorisasjon: en agent som representerer noen eller noe må ha autorisasjon til å representere en bruker.

Tillatelse: en agent må også ha tillatelse til å utføre forskjellige operasjoner.

Disse begrepene danner grunnlaget for en konseptuell modell av hvordan mobile agenter fungerer. Figur 3.1 viser disse konseptene. I figuren er det tegnet inn steder, reiser, møter og forbindelser.

Figuren viser også hvordan agentsystemet er et lag over operativsystemet og maskinen (merket *Host system* på figuren). Det figuren ikke viser er hvordan forholdet mellom applikasjonen som bruker mobile agenter og agentsystemet er. I praksis vil et agentsystem være nært knyttet til en applikasjon.



Figur 3.1: En konseptuell modell av hvordan mobile agenter fungerer (fra [GHN+97]).

For å beskrive de forskjellige tjenestene eller funksjonene som bør være i et agentsystem, har man funnet fram til seks modeller [GHN+97]:

Agentmodellen: (*agent*) definerer den interne strukturen til agenten. Dette innebærer regler for autonomitet, læring og samarbeid med andre agenter. I tillegg inneholder denne modellen regler for reaktivitet. Agenten må også ha en oppgave eller et mål (pro-aktivitet).

Livssyklusmodellen: (*life cycle*) gjør rede for de forskjellige tilstandene i løpet av en agents liv samt hendelsene som får agenten til å skifte tilstand. Det er to hovedmåter å beskrive livssyklusen til en agent på:

- I modellen for persistente prosesser har man de tre tilstandene *startet*, *stoppet* og *frosset*. Denne modellen forutsetter at agenten støtter migrering av kjøretilstand og brukes derfor i systemer som støtter sterke agenter.
- Den oppgavebaserte modellen brukes i agentsystemer som anvender svake agenter. Agenten må starte på nytt hver gang den kommer til en ny maskin, og de forskjellige oppgavene kan være implementert som funksjoner. Agenten bestemmer hvilke oppgaver den skal gjøre ut fra forskjellige betingelser på den lokale maskinen.

I prototypen i kapittel 4 brukes en oppgavebasert modell. Dette er på grunn av at agentsystemet Voyager ikke støtter migrering av tilstand.

Beregningsmodellen: (*computational*) definerer hva som skjer under kjøringen. Denne modellen bestemmer hva som er tillatte operasjoner for en agent, og fungerer som kjernen i agenten. Gjennom denne modellen får man tilgang til de andre funksjonene definert av de andre modellene.

Sikkerhetsmodellen: (*security*) spesifiserer forskjellige sikkerhetstiltak for å beskytte agenten og dens omgivelser. Her kan problemet deles opp i to hovedgrupper:

1. å beskytte maskiner fra infiserte agenter (for eksempel agenter med virus), og
2. å beskytte agenter fra infiserte maskiner.

Kommunikasjonsmodellen: (*communication*) definerer hvordan agenter kan kommunisere med andre agenter, med den lokale maskinen og andre tjenester.

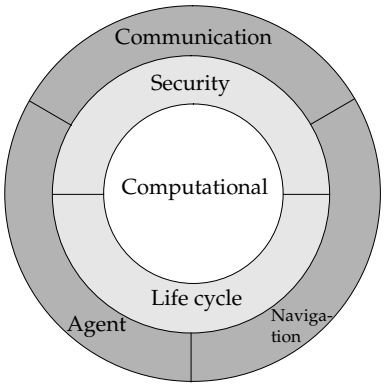
Navigasjonsmodellen: (*navigation*) definerer hvordan agenten skal finne fram til de forskjellige stedene i et distribuert system. Her er det nødvendig med forskjellige navneskjemaer slik at det er mulig å identifisere agenter, steder og tjenester.

En implementasjon av disse modellene vil utgjøre et agentsystem. Forholdet mellom disse modellene kan tegnes opp som på figur 3.2. Figuren viser beregningsmodellen som kjernen i agentsystemet. Beregningsmodellen er den viktigste modellen etterfulgt av modellene for sikkerhet og livssyklus. Inndelingen i figuren gir en oversikt over hvilken funksjonalitet som er den viktigste i et agentsystem.

3.4.1 Agentsystemer og mellomvare

En måte å opprette en kommunikasjonslink på er via *sockets* [Ste98]. En *socket* betegner entydig et endepunkt i en forbindelse mellom to prosesser. For å opprette en slik *socket* må den defineres, den må bindes til en adresse og den må stilles i lyttestilling. Alt dette kan programmers i C eller et annet passende språk.

Socket er eksempel på lavnivå programmering som krever at programmerer vet en hel del om de underliggende detaljene i nettverket og operativsystemet. I store distribuerte systemer er det ønskelig med et høyere abstraksjonsnivå slik at en del av disse detaljene blir skjult for programmerer. Det



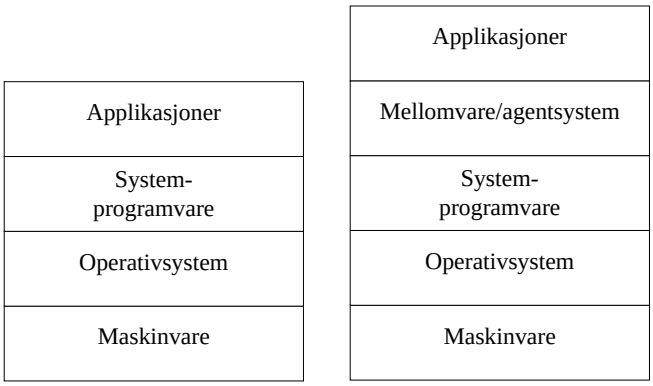
Figur 3.2: Forholdet mellom de forskjellige modellene i en mobil agent (fra [GHN+97]).

kan da bli lettere å utvikle distribuerte applikasjoner. Ut fra dette ønsket har det kommet en rekke produkter som betegnes som *mellomvare* eller *middleware*. Mellomvare kan defineres slik [Uma97, side 19]:

> *Middleware is a set of common business-unaware services that enable applications and end users to interact with each other across a network. In essence, middleware is the software that resides above the network and below the businessaware application software.*

Denne definisjonen legger vekt på at forretningslogikk ikke hører til i mellomvaren. Med forretningslogikk menes for eksempel administrasjon av dokumenter, e-post eller det som er hovedoppgaven til systemet i en bedrift. I prototypen i kapittel 4 er forretningslogikken i systemet å administrere romreservasjoner. Mellomvaren er generell og inneholder ikke støtte for spesielle forretningsfunksjoner. Målet er at mellomvaren skal skjule detaljer fra nettverket og operativsystemet. Dette kan gjøre det lettere å utvikle distribuerte applikasjoner fordi programmerer kan fokusere på å implementere forretningslogikken. Figur 3.3(b) viser hvor mellomvare befinner seg i forhold til en applikasjon og operativsystemet. Figur 3.3(a) viser en tradisjonell applikasjon som benytter systemprogramvaren på en maskin direkte.

I en artikkel av Philip A. Berstein [Ber96] blir mellomvare sett på som løsningen på heterogenitet mellom forskjellige maskiner og plattformer. Bedrifter og organisasjoner har hatt store problemer med å koble sammen sine interne nettverk på grunn av heterogenitet og manglende standarder. Dette fører igjen til problemer når bedriften skal bruke nettverket og applikasjonene som et internt informasjonsverktøy ovenfor ansatte og kunder. Som et svar på disse problemene har databransjen utviklet programvare som bruker standard protokoller og standard programmeringsgrensesnitt.



(a) Tradisjonell inndeling (b) Bruk av mellomvare

Figur 3.3: Mellomvare skjuler en del av kompleksiteten ved det underliggende systemet.

Bernstein kaller dette for mellomvaretjenester.

I følge Bernstein trenger programmerere høynivå grensesnitt som skjuler kompleksiteten i nettverk og protokoller. Dette gir utvikleren muligheten til å fokusere på applikasjonen istedenfor detaljene. Mellomvare kan også ses på som gjenbruk av komponenter eller biblioteker (se også side 20 i kapittel 2). Eksempler på mellomvare er implementasjoner av CORBA[2] og Java RMI [OH98].

Siden flere og flere distribuerte applikasjoner blir utviklet med mellomvare, vil også kravene til et distribuert system avhenge av hva mellomvaren kan tilby av tjenester. Det er derfor viktig å kartlegge hva slags funksjonalitet mellomvare kan tilby.

Et agentsystem kan også betraktes som en type mellomvare, men agentsystemet brukes da til å implementere agenter. Heller ikke et agentsystem har støtte for spesiell forretningslogikk. Det er applikasjonen som tar i bruk agentsystemet som implementerer forretningslogikken.

3.5 Mobile agenter og distribuerte systemer

Et distribuert system ble i innledningen forklart som en samling av datamaskiner koblet sammen i et nettverk som samarbeider om å tilby en tjeneste. En mer presis definisjon finnes for eksempel i [BS98, side 4]:

A distributed system is a system designed to support the develop-

[2]CORBA står for *Common Object Request Broker Architecture*.

> *ment of applications and services which can exploit a physical architecture consisting of multiple, autonomous processing elements that do not share primary memory but cooperate by sendig asynchronous messages over a communications network.*

Distribuerte systemer er en generell betegnelse på en samling maskiner som er koblet sammen for å utføre en tjeneste. Et eksempel på et distribuert system er et nettverk av minibanker. Agentsystemer som samarbeider er også et eksempel på et distribuert system.

Definisjonen over forklarer generelt hva et distribuert system er. Et distribuert system består av en samling autonome maskiner. En *autonom maskin* vil si at en maskin består av en prosessor og minne som kun denne maskinen kan bruke. *Asynkron meldingsutveksling* betyr at vi ikke kan anta noe om hvor lang tid det tar å få tilbake svar fra en forespørsel. Kommunikasjonen mellom maskinene skjer ved hjelp av meldingsutveksling over et nettverk, og ikke ved hjelp av felles minne.

Utfordringen ved å kommunisere over et nettverk er at datapakker kan bli borte eller forsinket. Dette kan skyldes overbelastning i nettverket eller forstyrrelser på overføringsmediet. Forskjellige protokoller brukes for å dekke over mesteparten av feilene i ulike typer nettverk.

I boka *Object-oriented client/server Internet Environments* [Uma97, side 51] har man en tilsvarende definisjon på et distribuert system. I definisjonen kaller man et distribuert system for *distributed computing system*:

> *A **distributed computing system (DCS)** is a collection of autonomous computers interconnected through a communication network to achieve business functions. Technically, the computers do not share main memory so that the information cannot be transferred through global variables. The information (knowledge) between the computers is exchanged only through messages over a network.*

Denne definisjonen legger også vekt på at målet med et distribuert system er å oppnå eller støtte en forretningsfunksjon i en organisasjon eller en bedrift. Systemet i seg selv inneholder ingen forretningsfunksjon.

Fordelen ved å ta i bruk et distribuert system blir nevnt i [BS98]. Et distribuert system gjør det mulig å dele ressurser som skrivere og annen maskinvare mellom maskiner. Et distribuert system gjør det også mulig å dele programvare og informasjon.

En annen viktig motivasjonsfaktor for å ta i bruk et distribuert system er at organisasjoner ofte er spredt over store geografiske områder. Et distribuert system kan derfor brukes til å opprettholde kommunikasjonen mellom forskjellige deler av en organisasjon.

Anvendelsesområdet for mobile agenter er i distribuerte systemer. Et agentsystem som skal støtte mobile agenter må derfor ta hensyn til spesielle egenskaper som for eksempel sikkerhet og samtidighet.

3.5.1 Egenskaper ved distribuerte systemer

Et distribuert system har mange egenskaper som også er viktige ved bruk av mobile agenter. I boka *Distributed Systems – Concepts and Design* [CDK94] trekker forfatterne fram disse egenskapene:

Ressursdeling: dette vil si at maskinvare, programvare og felles data kan deles mellom maskiner. Ved ressursdeling er det viktig å ha en enhet som administrerer ressursene. Oppgavene til en administrator kan være å se til at andre overholder regler for bruk av ressurser.

Åpenhet: vil si at maskin- og programvare er åpne for utvidelser og forandringer. To viktige begreper for å beskrive åpenhet er *interoperabilitet* og *portabilitet* [BS98, side 9]. Interoperabilitet betyr at distribuerte systemer fra forskjellige leverandører skal kunne virke sammen. For eksempel skal to forskjellige implementasjoner av CORBA kunne brukes sammen. Med portabilitet menes at programvare som bruker mellomvare skal kunne kjøre over ulike plattformer fra forskjellige leverandører. For eksempel bør det være slik at en applikasjon utviklet med Voyager [3] også lar seg bruke med ORBacus[4] som mellomvare. For å få til dette er det viktig å standardisere grensesnittene. I tillegg er det viktig å ha en arkitektur hvor komponentene i seg selv er åpne. Fordelene med dette er at interoperabilitet og portabilitet gjelder for alle komponentene i en arkitektur.

Samtidighet: er et resultat av at flere maskiner jobber i parallell. Et godt eksempel er et filsystem. Flere brukere kan skrive til samme fil samtidig. Tilgangen til filen må synkroniseres slik at innholdet blir konsistent. For å kontrollere samtidighet bruker man forskjellige låsemekanismer. Samtidighet må implementeres på en slik måte at fordelene med å prosessere i parallell ikke forsvinner.

Skalerbarhet: er muligheten til å utvide systemet. Skalerbarhet har nøye sammenheng med hvordan systemet er designet og implementert. Derfor er bruk av åpne standarder viktig for å støtte skalering. For

[3] Voyager er en CORBA implementasjon fra ObjectSpace. Voyager brukes også til å implementere prototypen i kapittel 4. Se `http://www.objectspace.com` (1. november 1999) for mer informasjon.

[4] ORBacus er en CORBA implementasjon fra Object Oriented Concepts. Se `http://www.ooc.com` (1. november 1999) for mer informasjon.

programvaren er det et mål at systemet skal kunne skalere uten store endringer. Maskinvare og nettverk setter også begrensninger på hvordan et system kan utvides. Videre er det viktig å bruke forskjellige former for replikering av tjenester for å fjerne flaskehalser i systemet. Bruk av replikering kan igjen føre til inkonsistens.

Feiltoleranse: vil si at et system klarer å håndtere feil som oppstår i maskin- og programvare. Vanlige teknikker for å bedre feiltoleransen er å utvide systemet med flere reservekomponenter. På denne måten gjør man systemet uavhengig av en enkelt komponent, men dette innfører nye problemer med at data kan bli inkonsistente.

Distribuerte systemer kan ha større feiltoleranse enn sentraliserte systemer. Et distribuert system kan designes slik at det er mulig å bruke systemet selv om en tjener går ned. Igjen er det kravene fra applikasjonen som er avgjørende for hvor mange feil et system kan tåle. En applikasjon for flykontroll har strenge krav til at systemet skal fungere selv om noe av maskinvaren eller programvaren slutter å fungere.

Transparens: skjuler forskjellige nivåer av systemet for utvikler og sluttbruker. For eksempel betyr replikeringstransparens at sluttbruker blir skånet for opplysninger om hvor filer, objekter eller tjenere finnes. Det finnes sju andre typer transparens definert i *Reference Model for Open Distributed Processing* (RM-ODP) [ISO95]. Formålet med transparens er å dele opp systemet i håndterlige deler og skjule de delene som er uvesentlige.

En av konseptene ved mobile agenter er at de utnytter ressursene på maskinen de er på. Programmerer bør derfor vite om adresser og tjenester på forskjellige maskiner. Transparens som gjelder skjuling av lokasjon blir derfor brutt ved programmering av mobile agenter.

3.5.2 Utfordringer ved distribuerte systemer

Utfordringer fra tradisjonelle distribuerte systemer gjelder også for agentsystemer og mellomvare [WWWK97]. Dette gjelder for eksempel hvordan man skal navngi alle entiteter i et system eller kontrollere samtidighet.

I de følgende avsnittene blir fire at de viktigste utfordringene ved distribuerte systemer forklart:

- navngiving,
- kommunikasjon,
- kontroll av samtidighet og konsistens, og
- sikkerhet

Navngiving

Ressursdeling i distribuerte systemer gjør det nødvendig med navn på de forskjellige delene av et system. Eksempler på enheter som trenger navn er maskiner, brukere, tjenere og prosesser. Det er også nødvendig med systemer som kan brukes til å finne en adresse fra et navn og omvendt [CDK94, side 256]:

- En navnetjener kan sammenlignes med de hvite sidene i en telefonkatalog. I en navnetjener kan man finne adressen til en maskin ved å slå opp på navnet.
- En annen type tjeneste kalles *trader* [BS98, side 33]. Denne tjenesten kan sammenlignes med de gule sidene i en telefonbok. Ved å slå opp på en ønsket funksjon, kan en *trader* finne fram til en liste over tjenere som tilbyr den ønskede funksjonen.

En navnetjeneste kan ses på som en database med bindinger mellom navn og attributter. For eksempel vil navnet `ifi.uio.no` ha adresseattributtet `129.240.64.2` (som i dette tilfellet er en IP-adresse). Et navn kan ha flere attributter knyttet til seg. Dette er praktisk hvis navnetjenerne brukes til flere formål enn å oversette adresser.

En navnetjeneste må ha regler for hva som er lovlige navn. Generelt kan navnerommet (det vil si alle tillatte navn) være flatt eller hierarkisk. Flate navn er gjerne tilfeldig genererte, mens hierarkiske navn er satt sammen av for eksempel den geografiske lokasjonen. Et eksempel på flate navn er objektidentifikatorene som brukes i mellomvare som CORBA og Java RMI [Dol97]. Eksempel på hierarkiske navn er IP adresser. Videre kan et navnerom være endelig eller uendelig. I en artikkel av Roger M. Needham [Nee93, side 321] legges det vekt på at det er en god designregel å anta uendelig mange navn i et distribuert system. Det skal ikke være nødvendig å redesigne hele systemet på grunn av at bruksområdet er utvidet eller forandret. Problemet er at det er vanskelig å anslå det maksimale antall entiter som må ha adresser.

For å oppnå god ytelse og tilgjengelighet i et distribuert system er det viktig at navnetjeneren ikke blir en flaskehals. Ved å replikere navnetjeneren kan man unngå dette. I store globale navnetjenester (som for eksempel i Domain Name System [Moc87]) deler man opp databasen med bindinger slik at hvert administrativt domene har ansvar for sine navn. Hele navnetjenesten har struktur som et tre hvor nodene i treet er navnetjenere som administrerer de nærmeste undernodene. Fordelen med dette er at systemet blir mer robust. Selv om en navnetjener går ned vil resten av navnesystemet fungere.

Kommunikasjon

Et distribuert system består av samarbeidende prosesser som kommuniserer over et nettverk. Effektiviteten til nettverket er kritisk for ytelsen og påliteligheten til applikasjonene som bruker nettverket. Samtidig med høy effektivitet og ytelse er det ønskelig å bevare et høynivå grensesnitt for programmering slik at det blir lettere å utvikle distribuerte applikasjoner.

Tradisjonelt har man brukt lagdeling for å beskrive hvordan kommunikasjon fungerer. Hvert lag har sitt ansvarsområde og et veldefinert grensesnitt mot andre lag. På denne måten blir et underliggende lag tilbyder av tjenester for laget over.

Den lagdelte modellen er oversiktlig fordi den deler inn komplekse kommunikasjonsoppgaver i grupper. Ulempen med lagdeling er at hvert lag prosesserer dataene slik at forsinkelsen blir merkbar. Det er ikke bare i endesystemene at lagdeling brukes, men også i nettverket (det vil si i svitsjer, rutere og broer). For å korte ned forsinkelsen, bruker man i dag enklere teknikker. Disse teknikkene overfører mye av prosessering og konvertering av pakker til endesystemene. Hvis de forskjellige nodene i nettverket gjør minst mulig prosessering, kan hastigheten øke samlet sett.

Kontroll av samtidighet og konsistens

Samtidighet er en egenskap som gjør det mulig å utføre oppgaver i parallell. Et eksempel er et program som skal kompileres. Ved å utføre kompileringen på mange prosessorer samtidig kan man få en raskere kompilering enn om man skulle ha utført kompileringen på en prosessor.

Problemene med samtidighet er at forskjellige prosessorer kan prosessere samme data samtidig og dermed produsere et feil resultat. En måte å løse dette problemet på er å sette en lås på dataelementene slik at kun en av prosessene kan aksessere data om gangen. Låsing kan innføre nye problemer for eksempel ved at to prosesser som venter på hverandre kan komme i vranglås (det vil si at de venter på hverandre).

I boka *Open Distributed Systems* [Cro96] blir følgende trukket fram som utfordringer i samtidige systemer:

Skedulering: planlegging av når og hvordan samtidige oppgaver kan bruke delte ressurser som prosessor, I/O og minne.

Synkronisering: det å bestemme rekkefølgen av oppgaver i systemet.

Kommunikasjon: når to prosesser eller tråder er synkronisert vil de gjerne dele informasjon, men hvordan skal kommunikasjonen mellom dem foregå?

Konsistens handler om å opprettholde et riktig bilde av den delte tilstanden i et distribuert system. I store systemer er det nødvendig å replikere data. Grunnen til dette er at man vil øke tilgjengeligheten og minke aksesstiden til systemet.

Sikkerhet

Sikkerhet er viktig fordi mange organisasjoner har mye informasjon lagret i databaser som er tilgjengelig via nettverk. Et godt eksempel er det amerikanske forsvarsdepartementet som har viktige databaser koblet opp mot Internett. Et annet aktuelt eksempel, hvor sikkerhet er viktig, er betalingsformidling over Internett.

Sikkerhet i distribuerte systemer handler om å beskytte informasjon og ressurser mot inntrengere. I [Sta97, side 624] blir dette kalt *network security*. Dette innebærer å beskytte data som overføres mellom maskiner og garantere for at overføringen er ekte. Den viktigste teknikken for å garantere dette er kryptering.

Umar [Uma97, side 121] presenterer en mer detaljert liste over hva sikkerhet innebærer:

Autentisering: vil si å bevise at en person virkelig er den han eller hun utgir seg for å være. Ikke alle brukere skal ha adgang til alle tjenester i et system.

Sporbarhet: betyr at det skal være mulig å logge enhver bruk av ressurser i et system. En logg kan inneholde hvem som brukte ressursen og når dette skjedde.

Konfidensialitet: i et system med mange brukere er det viktig å holde informasjon privat. For eksempel bør e-post bare leses av mottaker og ingen andre.

Integritet: går ut på at systemet bare skal inneholde korrekte meldinger. Dette vil si at ingen kan forandre eller sende inn falsk informasjon i systemet.

***Non-repudiation*:** går på at det skal være mulig å bevise at en avsender virkelig har sendt en melding, og at en mottaker har mottatt meldingen. Dette er viktig for eksempel ved elektronisk betaling.

3.6 Forskjellige modeller for distribusjon

Dette avsnittet tar for seg forskjellige modeller for å organisere kommunikasjon mellom to eller flere prosesser. Den kanskje mest kjente modellen for kommunikasjon er klient/tjener modellen. I denne modellen er det tjeneren som tilbyr forskjellig funksjonalitet til klienten. Kommunikasjonen skjer ved at en klient sender meldinger til en tjener som igjen utfører en eller flere funksjoner. Resultatet blir returnert til klienten. En prosess tilbyr funksjoner, mens en annen prosess bruker dem.

I boka [Uma97, side 52] innfører man også filoverføringsmodellen og *peer-to-peer* modellen:

Filloverføringsmodell: kommunikasjonen mellom maskinene skjer ved at de sender filer til hverandre. Mange systemer bruker forsatt denne modellen. E-post er et eksempel.

***Peer-to-peer* modell:** prosesser kan spille rollen som både klient og tjener etter behov. Denne modellen brukes hvis det ikke er noe klart skille mellom tilbyder og bruker av tjenester.

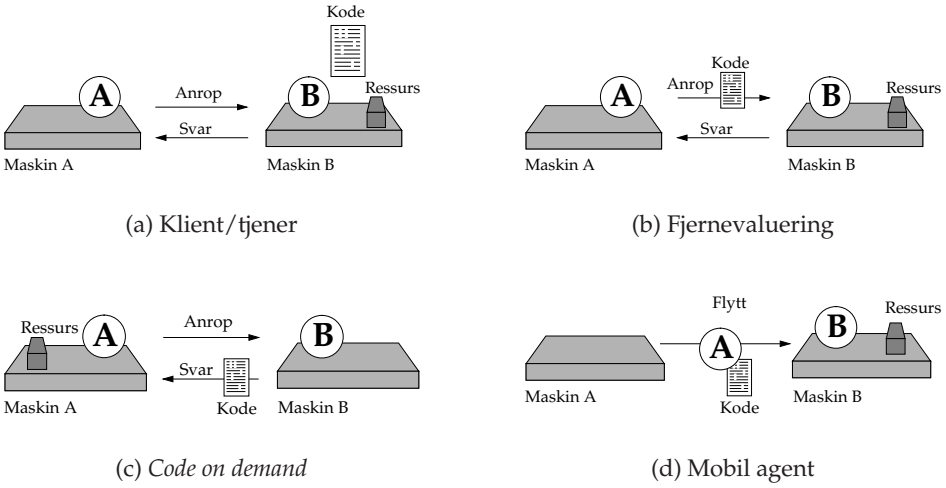
En annen inndeling av paradigmer finner vi i [FPV98, Vig98]. Denne inndelingen tar utgangspunkt i graden av mobiltet i forhold til kode, data og ressurser. Forfatteren kaller dette for paradigmer for mobil kode. Det blir presentert fire modeller:

Klient/tjener: en klient A gjør et anrop mot en tjener B. Svaret blir returnert tilbake til A. Klient/tjener-modellen er vist i figur 3.4(a).

Fjernevaluering: går ut på at en komponent A vet hvordan den skal gjøre en oppgave, men mangler ressursene (for eksempel regnekraft). Den sender derfor koden over til en annen prosess B som har ressurser. Prosess B beregner resultatet og returnerer svaret til A. Dette er vist i figur 3.4(b).

***Code on demand*:** prosess A vet ikke hvordan den skal gjøre en oppgave (fordi den ikke har koden), men den har ressursene. Den sender derfor ut en forespørsel om å få koden fra komponent B. Dette kalles *code on demand* og er vist i figur 3.4(c).

Mobil agent: prosess A vet hvordan den skal gjøre en oppgave, men A har ikke ressursene som skal til for å fullføre oppgaven. A gjør derfor deler av oppgaven i egen maskin, og forflytter seg deretter til B for å forsette eksekveringen der. Dette er vist i figur 3.4(d).



(a) Klient/tjener

(b) Fjernevaluering

(c) *Code on demand*

(d) Mobil agent

Figur 3.4: Forskjellige modeller for kommunikasjon (fra [Vig98]).

Disse fire paradigmene er ment å være alternativer til hvordan man designer kommunikasjonen i en applikasjon. Ved å analysere kommunikasjonsmønstrene kan man finne fram til hvilket paradigme som passer best for en applikasjon.

3.7 Oppsummering

Dette kapitlet har sett nærmere på hva mobile agenter er. Selve begrepet agent stammer fra forskning innen kunstig intelligens, mens teknologien til mobile agenter er en videreutvikling av forskning rundt prosessmigrering, fjernevaulering og mobile objekter.

Viktige egenskaper ved en mobil agent er at den er selvstendig (autonom), målrettet (pro-aktiv) og reaktiv. En av fordelene ved å bruke mobile agenter er at de kan forflytte seg over i forskjellige adresserom. På den måten kan en agent utnytte lokal kommunikasjon og få tilgang til interne tjenester i et program.

En mobil agent er egentlig et program. Teknisk baserer mobile agenter seg på tradisjonelle prinsipper for datakommunikasjon. Mobile agenter kan derfor ses på som en annen måte å organisere kommunikasjon på enn for eksempel klient/tjener arkitekturen.

Man er avhengig av et agentsystem for å bruke mobile agenter. Et agentsystem tilbyr agentene en kjøreomgivelse samt forskjellige nyttige tjenester. Funksjonaliteten til et agentsystem kan beskrives ved hjelp av forskjellige

modeller.

Mobile agenter og agentsystemer står ovenfor mange av de samme utfordringene som i distribuerte systemer. Siste del av dette kapitlet så derfor på egenskaper i distribuerte systemer. Et agentsystem kan skjule noen av utfordringene ved å abstrahere bort detaljer fra operativsystemet og nettverket. Dette ble kalt mellomvare.

Siste avsnitt sammenlignet mobile agenter med andre modeller for distribusjon. Inndelingen tok utgangspunkt i hvor kode og ressurser befinner seg i et nettverk. Ved å analysere en distribuert applikasjon kan man finne fram til den modellen som passer best.

Kapittel 4

Bruk av mobile agenter til vedlikehold

Dette kapitlet presenterer et eksempel og en prototype på hvordan mobile agenter kan brukes til vedlikehold i distribuerte systemer. Målet med prototypen er å vise hvordan nye funksjoner kan legges til dynamisk uten at sluttbruker behøver å installere ny programvare. Agenten skal ordne dette.

Foruten å beskrive hvordan prototypen fungerer, vil kapitlet også se nærmere på hvordan mobile agenter er implementert og hvordan arkitekturen er.

Prototypen belyser en del problemer i forbindelse med bruk av mobile agenter til vedlikehold. Disse problemene blir diskutert nærmere i neste kapittel.

4.1 Et eksempel på et distribuert system

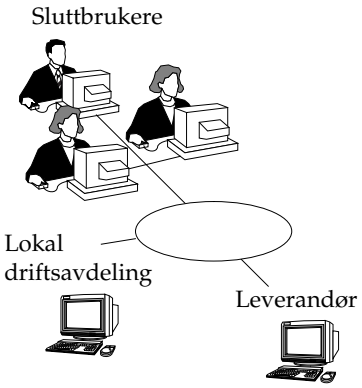
For å demonstrere hvordan mobile agenter kan utføre vedlikehold, har jeg laget en enkel prototype som har fått navnet MORP (*Multi-Organization Resource Planner*). Meningen med MORP er at programmet skal kunne brukes til å holde orden på ressurser mellom samarbeidende bedrifter eller organisasjoner. Ved å bruke MORP skal en ansatt fra en organisasjon kunne reservere rom som tilhører en annen organisasjon. Dette er praktisk i prosjekter hvor bedrifter samarbeider tett. Man kan tenke seg at MORP kan brukes til mer enn å bare reservere rom, men i prototypen vil det kun være støtte for dette. Eksempler på annen funksjonalitet kan være møteplanlegging eller støtte for gjennomføring av selve møtet.

Mobile agenter brukes til å utvide funksjonaliteten i MORP etter at systemet er tatt i bruk. Prototypen i seg selv vil ikke være brukbar som et selvstendig program, men gir nyttige erfaringer med hensyn til bruk av mobile agenter.

Figur 4.1 viser tre mulige brukergrupper i dette systemet: *sluttbrukere*, *driftsavdelingen* og *leverandøren* av systemet. Sluttbrukerne er de som bruker systemet i sitt daglige virke til å reservere rom. Driftsavdelingen har ansvar for lokalt vedlikehold, mens leverandøren har ansvar for videreutvikling og feilretting.

MORP er et distribuert system og består av flere klienter og en tjener. Typisk har sluttbruker tilgang til klienten, mens driftsavdelingen og muligens leverandøren har tilgang til tjeneren.

Formålet med prototypen er å vise hvordan mobile agenter kan brukes til vedlikehold. Det er derfor ikke lagt stor vekt på andre områder som for eksempel det å lage et bra brukergrensesnitt. Kildekoden, mer teknisk informasjon og instruksjon om hvordan applikasjonen startes finnes i vedlegg A.

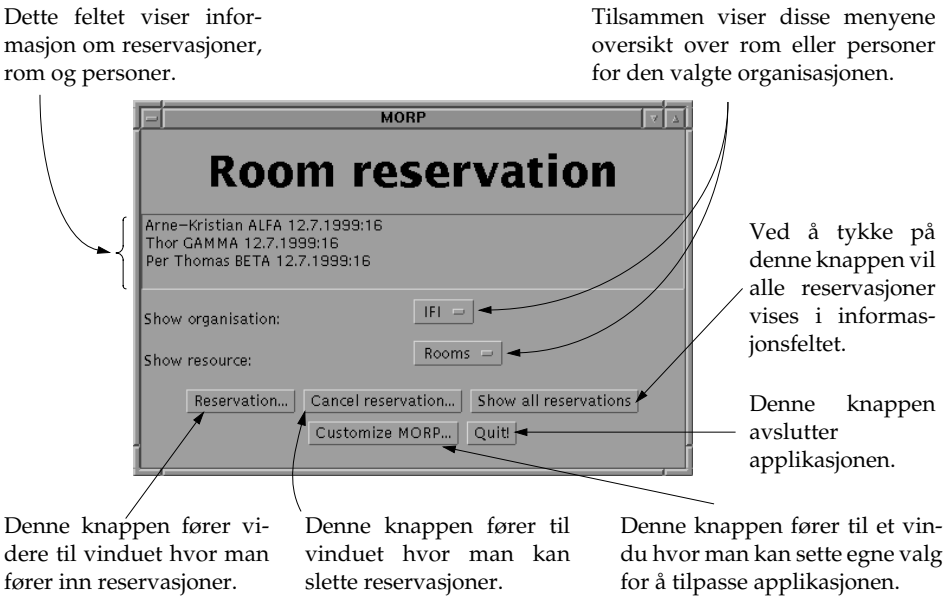


Figur 4.1: Figuren viser de forskjellige brukergruppene i systemet.

Eksemplet som følger er delt opp i to deler:

1. Første del forklarer hvordan MORP virker for en sluttbruker. Alle skjermbilder blir også forklart. Jeg vil også beskrive hvordan en bruker kan reservere et rom (scenario 1).

2. Andre del beskriver hvordan MORP kan oppgraderes med en ny funksjon. Den nye funksjonen gjør det mulig å legge til en saksliste eller agenda for hver reservasjon. En mobil agent brukes for å distribuere og legge til funksjonen. Dette eksemplet viser hvordan en driftsavdeling eller en produsent av programvaren automatisk kan oppgradere applikasjonen uten innblanding fra sluttbrukere (scenario 2).



Figur 4.2: Hovedskjermbildet for romreservasjon. En sluttbruker vil møte dette skjermbildet når han/hun starter programmet.

4.1.1 Hvordan virker systemet for en sluttbruker?

I en tenkt bedrift som benytter MORP vil ansatte ha klienten installert på sin maskin. De viktigste skjermbildene som sluttbruker ser er:

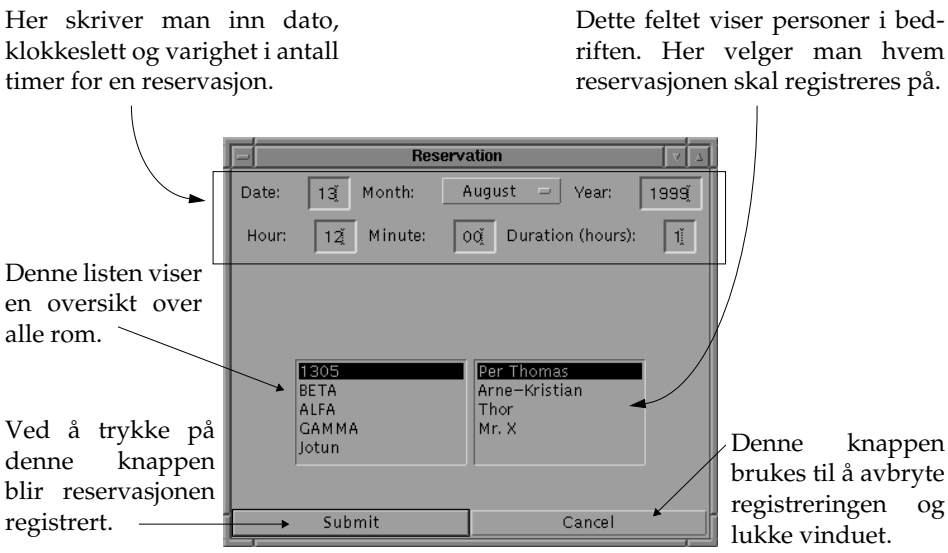
Hovedskjermbildet: dette er det første skjermbildet som vises når applikasjonen startes. Her har man oversikt over rom, ansatte og reservasjoner. Fra dette skjermbildet er det tilgang til de andre funksjonene i programmet. Hovedskjermbildet er vist i figur 4.2.

Reservasjonsbildet: i dette skjermbildet kan en bruker reservere et rom. Her velger man rom og tidspunkt for reservasjonen. Reservasjonsbildet er vist i figur 4.3.

Figurene viser hva de forskjellige knappene og menyene brukes til. I tillegg er det to andre skjermbilder for å slette reservasjoner samt konfigurere applikasjonen. Disse skjermbildene er ikke relevante for eksemplet, og de er derfor ikke beskrevet nærmere.

Scenario 1: Reservering av rom

For å foreta en romreservasjon må brukeren trykke på *Reservation...* knappen fra hovedskjermbildet. Man kommer da til reservasjonsbildet.



Figur 4.3: Skjermbildet for å registrere en eller flere romreservasjoner.

Her registreres informasjon om tidspunkt, varighet og hvilket rom reservasjonen gjelder. For å registrere reservasjonen trykker man *Submit*. Ved å trykke *Cancel* lukkes reservasjonsbildet.

4.1.2 Hvordan legge til en funksjon?

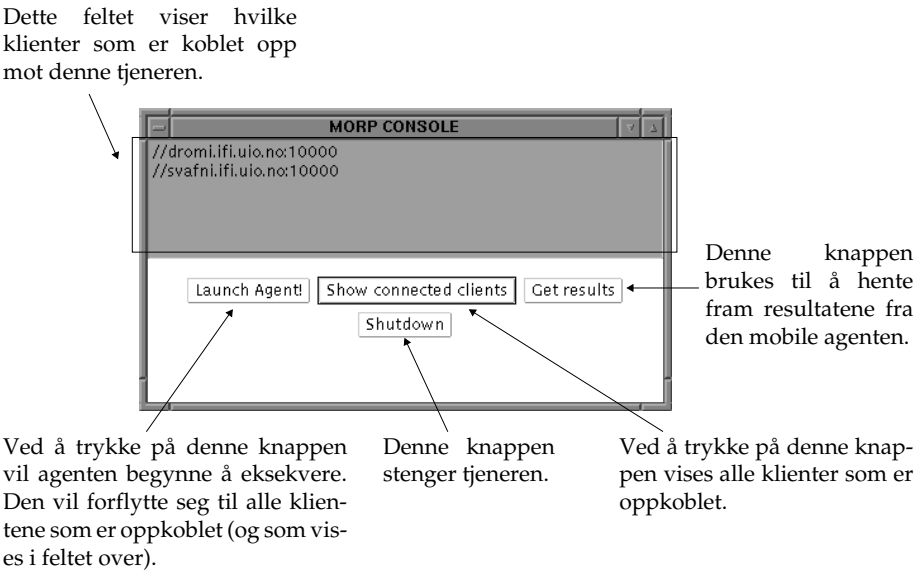
Forrige scenario var beskrevet ut fra sluttbruker og klientsiden av applikasjonen. På tjenersiden finnes det bare ett skjermbilde (vist i figur 4.4). I figuren er det forklart hva de forskjellige knappene brukes til.

Fra dette skjermbildet er det mulig å få en oversikt over hvilke maskiner (klienter) som er koblet opp mot denne tjeneren samt sende ut en agent for å utvide klientene med en ekstra funksjon.

Scenario 2: Å legge til en funksjon

I dette scenarioet blir en ekstra funksjon lagt til systemet. Denne funksjonen gjør det mulig å legge inn en saksliste (agenda) til hver reservasjon. I dette eksemplet kan man tenke seg at leverandøren har utviklet denne nye funksjonen. For å installere funksjonen brukes en mobil agent. Initiativet til å starte agenten kan gjøres via tjeneren.

Med utgangspunkt i skjermbildet for tjeneren (figur 4.4) vil en systemansvarlig få opp oversikt over klienter som er tilkoblet ved å trykke på knappen



Figur 4.4: Skjermbildet på tjenersiden.

merket *Show connected clients*. Det er disse klientene man kan sende agenten til.

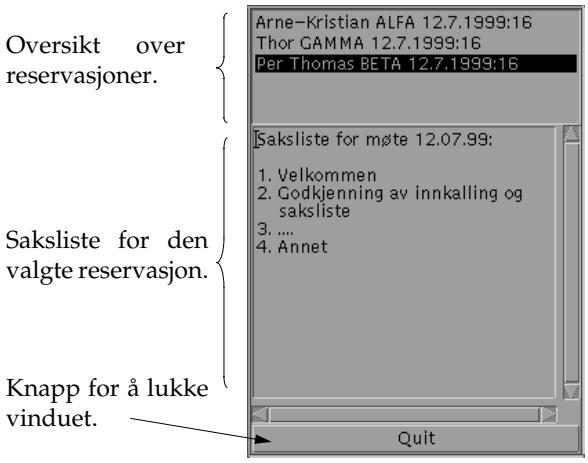
I dette eksemplet er en mobil agent kodet inn i tjeneren. I en mer fleksibel løsning kan agenten spesifiseres ved hjelp av klassenavnet slik at det er mulig å ha flere typer agenter. Den mobile agenten er programmert spesielt for å legge til støtte for saksliste.

Ved å trykke på knappen merket *Launch Agent* blir agenten overflyttet til alle oppkoblede klienter. Når agenten kommer til en klient vil den installere en ny knapp i hovedskjermbildet (figur 4.2). I tillegg til de seks knappene, er det kommet en ny knapp (*Add agenda*). Ved å trykke på denne knappen kommer man til skjermbildet for å legge inn sakslister for reservasjonene (vist i figur 4.5).

Etter at den nye funksjonen er installert, vil den mobile agenten forflytte seg til neste klient og gjøre det samme.

Hvordan ser koden til den mobile agenten ut?

For å implementere mobile agenter inn i MORP var det nødvendig med et agentsystem. Valget falt på Voyager fra ObjectSpace [Voy99] som blant annet tilbyr kommunikasjonstjenester og støtte for mobile agenter. I MORP er Voyager brukt til kommunikasjon mellom klient og tjener samt til å implementere mobile agenter.



Figur 4.5: Dialogboks for å skrive inn saksliste for forskjellige reservasjoner. Øverste liste viser en oversikt over forskjellige reservasjoner. Ved å klikke på denne listen vil den tilhørende sakslisten vises i tekstfeltet under. I tekstfeltet er det mulig å legge til eller forandre på eksisterende saksliste.

Ved å trykke på *Launch Agent*-knappen, blir følgende kode eksekvert:

```
1  agent = (IUpgradeAgent) Factory.create( UpgradeAgent.class.getName() );
2  agent.createTour( placeList.getItems () );
3  agent.go();
```

Første linje lager et nytt agentobjekt av typen `IUpgradeAgent`. Agentobjektet lages ved hjelp av en metode fra Voyager.

Den mobile agenten må vite hvilke maskiner den skal besøke. Dette gjøres i linje 2 ved hjelp av kall på metoden `createTour`. Denne metoden setter opp en intern liste i agenten over klienter den skal besøke. Adressen til en klient er angitt ved hjelp av en tekststreng på formen: `//<maskinnavn>:<portnummer>`.

Linje nummer 3 starter agenten ved å kalle på `go` metoden i agenten. Metoden `go` ser slik ut:

```
1   public void go()
2   {
3      String  nextPlace = null;
4      try
5      {
6         // set resource loader
7        URL url = new URL( "http://"
8        + MorpServer.getServerName()
9        + ":"
10       + MorpServer.getServerPort() );
11
12       URLResourceLoader loader = new URLResourceLoader( url );
13       Agent.of( this ). setResourceLoader( loader );
```

```
14          ...
```

Siden agenten skal flyttes rundt til forskjellige klienter, må den vite hvor den finner sine egne klasser. For at agenten skal kunne klare dette, må den bruke en annen metode for å laste klasser (*classloader*). Den nye *classloaderen* må klare å finne klasser over nettet. I dette tilfellet skal agenten laste klasser fra tjeneren. Metoden `setResourceLoader` brukes til å fortelle agenten hvor den skal finne klassene den trenger. Dette er vist i linje 13.

```
15          ...
16          // get next place to  visit
17          nextPlace = getNextPlace();
18
19          // move the agent
20          Agent.of( this ). moveTo( nextPlace, "doWork");
21       }
22       catch ( PlaceException pe )
23       {
24          System.err. println ( "Last location reached" );
25       }
26       catch( Exception e )
27       {
28          System.err. println ( "UpgradeAgent: "+ e );
29          e.printStackTrace();
30       }
31    }
```

Linje 15 til 21 viser hvordan agenten får adressen til neste klient samt hvordan den forflytter seg. Linje 20 bruker metoden `moveTo` fra Voyager. Som parametre tar denne metoden en destinasjon i form av en adresse og navnet på en metode som agenten skal kalle når den kommer fram til destinasjonen. Metoden `doWork` blir utført for hver klient, og agenten i MORP er således oppgavebasert (se også side 33). Det er i denne metoden at selve oppgraderingen av klienten skjer.

En forenklet utgave av metoden `doWork` ser slik ut:

```
1    public void doWork()
2    {
3       ...
4       MorpFrame.getInstance().addButton( new AddAgendaButton() );
5       go();
6    }
```

I linje 4 blir den nye knappen (`AddAgendaButton`) for saksliste lagt til i brukergrensesnittet. Knappen inneholder kode for hva som skal skje når den blir trykket på. Mer trenger ikke agenten å gjøre for å oppgradere klienten. Alle klasser som brukes i forbindelse med den nye funksjonen blir lastet ned over nettverket når de trengs. Metoden `addButton` i linje 4 er en generell metode for å legge til ny funksjonalitet.

Linje 5 inneholder et rekursivt kall på `go` i agenten. Dette er nødvendig

for å forflytte agenten til neste klient. Når alle klientene har blitt besøkt vil agenten få en *exception* og rekursjonen vil stoppe.

4.2 Arkitekturen til MORP

Utviklingen av MORP har vært gjennom tre faser:

1. Første fase la vekt på objektmodellen eller forretningsmodellen. Til dette brukte jeg et scenario for å reservere rom mellom bedrifter.
2. Andre fase fokuserte på distribusjon. Prototypen bruker en klient/tjener modell som kan utvides til en 3-lags arkitektur.
3. Tredje fase så på hvordan mobile agenter kunne brukes til vedlikehold. En del av oppgaven her var å finne ut hvordan et agentsystem skulle tas i bruk i den eksisterende koden.

Hver av disse fasene førte til forskjellige versjoner av prototypen. Applikasjonen fra første fase kjørte kun på en prosess og hadde ikke støtte for kommunikasjon. Formålet var å teste forretningsmodellen. Resultatet av andre fase var en klient/tjener applikasjon utviklet ved hjelp av ILU (*Inter-Language Unification*) [JSLJ99] fra Xerox[1]. Utfordringen her var å teste hvilke grensesnitt systemet måtte ha. Resultatet av siste fasen er MORP prototypen med mobile agenter slik den er presentert i dette kapitlet.

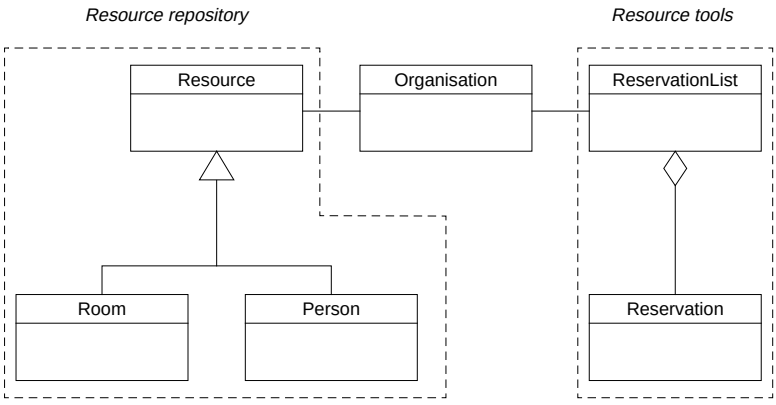
4.2.1 Objektmodellen

For å utvikle et system er det viktig å finne fram til de riktige objektene (og dermed klassene) som utgjør forretningsobjektene i systemet. I arbeidet med å finne fram til klasser har jeg brukt *use-cases*[FS97]. *Use-cases* er beskrivelser av oppgaver som en bruker kan gjøre i applikasjonen. Scenario 1 i dette kapitlet er et eksempel på et slikt *use-case*.

Ut fra flere *use-cases* har jeg satt opp et klassediagram som viser oversikt over hovedklassene i systemet (se figur 4.6). Klassene er gruppert i *ressurser* og *verktøy*. Ressurser representerer generelle ressurser, og er for eksempel rom og personer. Verktøy er de funksjonene som kan anvendes på ressursene. Et eksempel på et verktøy er en *reservasjon* fordi den binder et rom til en person i et tidsrom.

Figur 4.6 er et forenklet diagram og viser derfor ikke alle detaljene i implementasjonen. Dette er gjort for å få fram hovedtrekkene i systemet. De forskjellige klassene har disse rollene:

[1] `http://www.parc.xerox.com` (8. oktober 1999)



Figur 4.6: Oversikt over hovedklassene i MORP.

Organisation Denne klassen representerer en organisasjon. Formålet med denne klassen er blant annet å holde orden på hvilke ressurser som hører til hver enkelt organisasjon.

Resource Denne klassen representerer en generell ressurs. Klassen inneholder det som er felles for ressurser slik som navn på ressursen og hvilken organisasjon som er oppført som eier.

Person Klassen representerer en person eller en ansatt. Denne klassen arver fra *Resource* klassen og inneholder informasjon som er spesifikk for en person. Dette kan for eksempel være personnummer.

Room Klassen representerer et møterom i en organisasjon. Denne klassen arver egenskaper fra *Resource* klassen. Klassen inneholder også informasjon om hvor mange personer det er plass til i rommet.

ReservationList Denne klassen representerer en liste over alle reservasjoner som er gjort i systemet.

Reservation Denne klassen representerer en reservasjon. En reservasjon er en binding mellom flere ressurser og et tidsintervall. Klassene for tidsintervall er ikke tegnet inn.

4.2.2 Distribusjon og grensesnitt

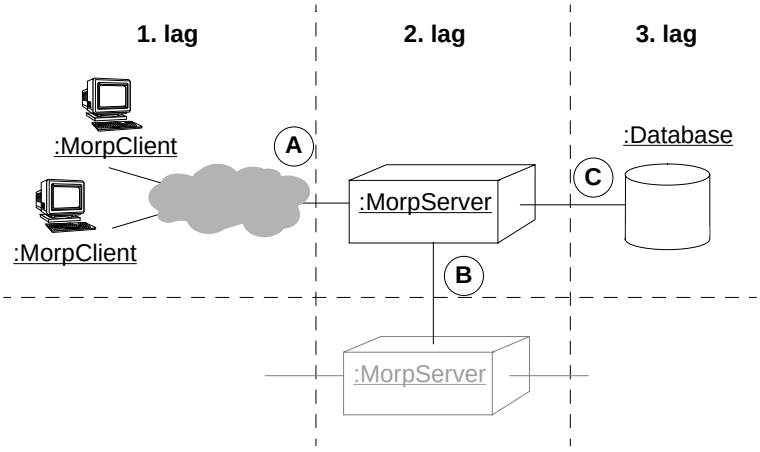
MORP bruker en trelags arkitektur som vist på figur 4.7. Hensikten med en slik arkitektur er å skille mellom presentasjonsobjektene, forretningslogikken og lagringsstrukturen [Uma97]. På figuren vises dette som lag 1, 2 og 3:

- Lag 1 er brukergrensesnittet slik det ser ut for sluttbrukerne.

- Lag 2 representerer tjeneren. Denne inneholder alle regler for hva som er lov å gjøre i systemet. Tjeneren har også grensesnitt mot klientene (merket `A`), mot en database (merket `C`) og mot andre tjenere (merket `B`).
- Lag 3 er en database eller en annen form for lagringsteknikk. All kommunikasjon med databasen går igjennom tjeneren (lag 2). I denne databasen er det mulig å lagre reservasjonene.

I et distribuert system er det viktig at grensesnittet mellom de forskjellige komponentene er bra definert. Det er gjennom grensesnittet at all kommunikasjon skjer.

Ut fra figur 4.7 ser vi at tjeneren bør støtte tre grensesnitt. Grensesnitt `A` brukes til å kommunisere med klientene. Grensesnitt `B` brukes til å kommunisere mellom tjenere. Hensikten med dette er for eksempel å tillate reservasjoner på tvers av organisasjoner. Grensesnitt `C` brukes til å kommunisere med en database. Det er kun grensesnitt `A` som er implementert i prototypen.

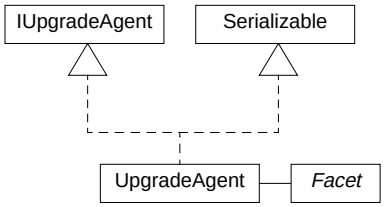


Figur 4.7: Trelags arkitekturen i MORP.

En måte å spesifisere grensesnittet på er å bruke `interface` nøkkelordet i Java. Denne teknikken er brukt i MORP. Klassene som spesifiserer grensesnittene i MORP finnes i vedlegg B.

I Voyager må alle klasser som skal være grensesnitt implementere `java.io.Serializable` i tillegg til å implementere sitt eget grensesnitt. Slike klasser kalles *remote enabled classes* i Voyager. Alle slike klasser kan brukes mellom en klient og en tjener.

Første gang en klient kaller en metode på tjeneren, må det genereres såkalte proxy klasser. Formålet med disse klassene er å skjule for klienten at meto-



Figur 4.8: Oversikt over `UpgradeAgent` med agent-*facet*.

den den kaller finnes i et annet adresserom. For en programmerer ser det ut som om klienten kaller en metode direkte på tjeneren, men egentlig er det en metode i proxy klassen som blir kalt. Denne metoden vil igjen sørge for kommunikasjonen mot tjeneren. Hvis et svar skal returneres tilbake til klienten, er det en proxy klasse på tjenersiden som overfører svaret tilbake til proxy klassen på klientsiden.

I fase to av MORP ble ILU brukt som et rammeverk for kommunikasjon mellom tjener og klienter. ILU kan brukes som et generelt rammeverk for å utvikle distribuerte systemer, men inneholder ikke støtte for å implementere mobile agenter. ILU støtter et spesifikasjonsspråk som heter ISL (*Interface Specification Language*). ISL er et deklarativt språk og brukes til å spesifisere hvordan grensesnittet skal se ut. I MORP ble grensesnittene spesifisert ved hjelp av ISL. I motsetning til i Voyager måtte alle proxy klasser genereres manuelt før kompilering.

Ved å bruke ISL er det mulig å koble sammen en versjon av MORP skrevet i for eksempel Lisp sammen med en versjon skrevet i Java. Et viktig poeng med spesifikasjonsspråk som ISL er at dette er egne språk med en egen syntaks. Grensesnittet (skrevet i ISL) og applikasjonen er derfor adskilt. Dette er ikke tilfellet ved bruk av Voyager hvor Java også brukes som spesifikasjonsspråk for grensesnittene. Konsekvensene av dette blir diskutert i neste kapittel (se avsnitt 5.1.2).

4.2.3 Mobile agenter

Klassen `UpgradeAgent` implementerer den mobile agenten som brukes i MORP. `UpgradeAgent` er en vanlig klasse i Java, men den må implementere sitt eget grensesnitt (`IUpgradeAgent`) og `java.io.Serializable`. Grunnen til dette er at metodene som er definert i grensesnittet kan kalles fra forskjellige prosesser. Serialiseringen gjør at objektet og alle tilhørende objekter kan sendes over nettverket.

I Voyager er egenskapene til en agent implementert via såkalte *facets*. *Facets* er enkelt forklart objekter som dynamisk kan knyttes til et objekt. Dette er

`moveTo`	Forflytter agenten til en annen prosess.
`setAutonomous`	Brukes til å bestemme om agenten skal fjernes fra minnet.
`isAutonomous`	Brukes til å finne ut om en agent kan fjernes fra minnet.
`getHome`	Returnerer adressen til prosessen hvor agenten ble startet.
`preDeparture`	Brukes av agenten til å utføre oppgaver før den forflytter seg.
`preArrival`	Brukes av agenten til å utføre oppgaver før hele agenten er forflyttet.
`postArrival`	Brukes av agenten til å utføre oppgaver etter at agenten har forflyttet seg til en ny destinasjon.
`postDeparture`	Brukes til å rydde opp på startstedet til agenten.

Tabell 4.1: Forskjellige metoder for å støtte mobile agenter i Voyager.

en enkel måte å utvide funksjonaliteten til et objekt. For å knytte en agent-*facet* til et annet objekt, kaller man `Agent.of()` metoden. Dette er vist på figur 4.8.

Voyager støtter også flere metoder for å signalisere når en mobil agent kommer til eller forlater en prosess. Disse er oppsummert i tabell 4.1. Den viktigste metoden er `moveTo` som gjør at agenten forflytter seg.

I kapittel 3 ble fem forskjellige modeller for mobile agenter presentert (se side 33). Funksjonaliteten i disse modellene støttes også i Voyager, men det er ingen klare begreper om disse i Voyager. Ansvaret for å implementere modellen er fordelt mellom agentsystemet Voyager og MORP. For eksempel er mesteparten av det som kalles beregningsmodellen implementert i MORP.

4.3 Oppsummering

Dette kapitlet har presentert prototypen MORP. Hensikten med prototypen er å vise hvordan mobile agenter kan brukes til å vedlikeholde klientene i et distribuert system.

I prototypen ble klientene utvidet med en funksjon for å legge til en saksliste for hver reservasjon. Dette ble beskrevet i scenario 2. Fordelen med å

bruke en mobil agent er at agenten kan forflyttes mellom ulike adresserom. Agenten har derfor tilgang til lokale metoder som normalt bare ville ha vært tilgjengelig for klienten selv.

Kapitlet har også tatt for seg arkitekturen til MORP. Viktige steg under utviklingen var å finne fram til objekter i systemet (steg 1), finne fram til en passende modell for distribusjon (steg 2) og ta i bruk mobile agenter (steg 3). Hovedtrekk i arkitekturen er at MORP er satt opp etter en trelags arkitektur med grensesnitt mot klienter, andre tjenere og mot databasen.

Prototypen bruker agentsystemet Voyager til å implementere mobile agenter. Neste kapittel inneholder en evaluering av prototypen.

Kapittel 5

Evaluering av MORP

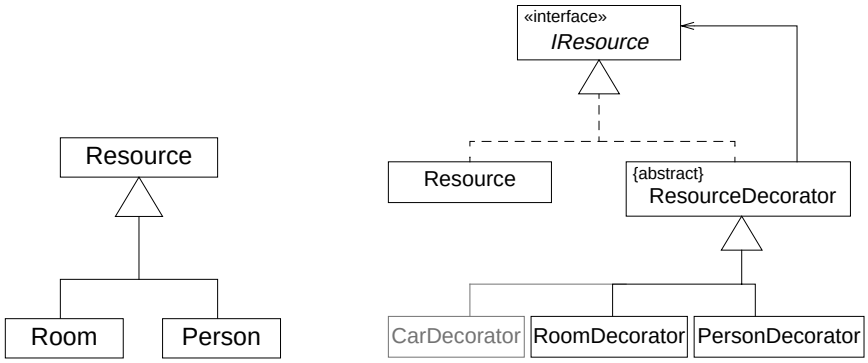
I dette kapitlet evaluerer jeg forskjellige sider ved prototypen MORP. Kapitlet begynner med å se på forskjellige deler av *prosessen* benyttet ved utvikling av MORP. Som nevnt i kapittel 4 utviklet MORP seg gjennom tre steg: design av objektmodell, valg av modell for distribusjon og bruk av mobile agenter. Her vil jeg se nærmere på hva som kunne bli bedre i hver av disse stegene.

Den andre delen av dette kapitlet evaluerer *produktet* (det vil si prototypen). Dette innebærer å se på hvilke typer vedlikehold som er mulig i MORP, og om bruk av mobile agenter gir noen fordeler med hensyn til vedlikeholdbarheten på systemet.

5.1 Evaluering av prosessen

Prototypen i denne oppgaven har stort sett vært basert på eksperimentering. Dette var nødvendig for å finne ut av hva systemet skulle gjøre. Kravene til systemet har derfor blitt klarere etter hvert. Den eksperimentelle fremgangsmåten førte ikke til noe spesielt valg av systemutviklingsmetode. Metodene under utvikling har derfor vært en blanding av *code-and-fix* og prototyping (se side 14). Konsekvensene av dette var at vedlikehold ikke ble planlagt fra begynnelsen av prosjektet. Dette er i motsetning til teorien som ble presentert i figur 2.1 på side 18 som sier at vedlikehold bør vektlegges spesielt i startfasen.

Dette er et generelt problem fordi det først og fremst fokuseres på hvordan applikasjonen skal virke før vedlikehold blir planlagt. En aktivitet som jeg utførte under utvikling av MORP var generalisering av objektmodellen. Dette kan være en måte å øke vedlikeholdbarheten på. Etter min mening kan det være lettere å legge til nye klasser i en generell modell.



(a) MORP subklassing (b) *Decorator* mønsteret

Figur 5.1: Forskjellge alternativer til subklassing.

5.1.1 Objektmodellen

Objektmodellen gir en oversikt over hvilken klasser systemet bør bestå av for å løse sine oppgaver. I tilfellet med MORP var oppgaven til systemet å administrere reservasjoner.

For å utvikle et fleksibelt system, var det viktig å lage en generell objektmodell. Dette ble gjort ved å innføre klasser som *ressurser* og *verktøy* (beskrevet nærmere på side 54). *Ressurser* er en generalisering av personer og rom, mens *verktøy* er en generalisering av reservasjonsbegrepet. Dette skulle gjøre det mulig å legge til støtte for nye ressurser (for eksempel biler) og nye verktøy (for eksempel det å leie en bil).

Måten dette ble implementert på var via subklasser som vist på figur 5.1(a). Min erfaring er at subklassing er en teknikk som er for statisk fordi nye klasser ikke kan legges til dynamisk (det vil si når systemet kjører).

En måte å løse dette på er å bruke *Decorator* mønsteret[1] beskrevet i [GHJV96, side 175]. Mønsteret går ut på at man har et grensesnitt (`IResource`) som spesifiserer metodene som alle ressurser må ha. Dette kan for eksempel være en metode for å få tak i navnet på ressursen. Klassen `Resource` implementerer `IResource` grensesnittet. Den abstrakte klassen `ResourceDecorator` kan inneholde en blanding av grensesnitt og implementasjon. Denne klassen har også en referanse til grensesnittet `Resource`. Klassene `CarDecorator`, `RoomDecorator` og `PersonDecorator` er eksempler på implementasjoner av den abstrakte klassen `ResourceDecorator`. Dette er vist i figur 5.1(b).

[1] Mønster er min oversettelse av *pattern*.

Dette mønstret gjør det mulig å legge til (eller dekorere) et annet objekt med nye egenskaper. Klassene som utgjør den nye funksjonaliteten kan legges til dynamisk. *Facets* som brukes i Voyager er en lignende teknikk (se side 57).

Faren ved å generalisere for mye er at objektmodellen kan bli vanskelig å forstå. Begrepene *ressurs* og *verktøy* er mer generelle en henholdsvis begrepene *rom* og *reservasjon*. I dokumentasjonen bør slike generelle begreper forklares med konkrete eksempler.

5.1.2 Distribusjon i MORP

Den andre fasen fokuserte på hvordan oppdelingen av systemet skulle være. Valget falt på en 3-lags arkitektur som beskrevet på side 55. Etter min mening er denne arkitekturen fornuftig fordi den skjuler databasen fra klienten. På den måten er det mulig å skifte databasen uten å forandre klienten. En annen fordel er at mest mulig av applikasjonslogikken er implementert i tjeneren, mens brukergrensesnittet er implementert i klienten.

Erfaringene fra ILU og fra Voyager viser to forskjellige måter å designe grensesnittene på:

1. Grensesnittet i ILU ble spesifisert ved hjelp av ISL. Vi forsøkte å spesifisere et enkelt grensesnitt med tekster som parametre og returverdier. Dette var på grunn av at alle typer vi brukte i grensesnittet også måtte deklareres i grensesnittet.

2. Grensesnittet i Voyager ble spesifisert ved hjelp av `interface` nøkkelordet i Java. Her ble det ikke tatt hensyn til at grensesnittet skulle være enkelt. Objekter ble brukt som parametre og returverdier.

I det første forsøket ble grensesnittet uavhengig av resten av programmet. På grunn av at alle typer som ble brukt i grensesnittet også måtte deklareres der, ble det tungvint å kommunisere mellom klient og tjener.

I det andre tilfellet ble mesteparten av objektmodellen i systemet trukket ut i grensesnittet. I prototypen fungerte dette greit fordi man kunne spesifisere grensesnittet i Java ved hjelp av `interface` nøkkelordet.

5.1.3 Bruk av mobile agenter

Implementeringen av mobile agenter skjedde etter at systemet fungerte som et enkelt romreservasjonsprogram. Agenten i MORP legger til støtte for agenda. Dette var hardkodet inn i agenten. Et mer fleksibelt system ville ha tillatt å konfigurere agenten før den ble sendt avgårde.

For å få den mobile agenten til å utvide klientene var det nødvendig med noen forandringer:

- Et kontrollvindu for agenten ble programmert. Vinduet inneholder oversikt over klientene i systemet samt en knapp for å starte agenten. Skjermbildet vises i figur 4.4 på side 51.
- To nye metoder ble lagt til i klassen `MorpFrame`. Dette er klassen som implementerer hovedskjermbildet (se figur 4.2 på side 49). Den ene metoden brukes til å få en referanse til objektet som representerer hovedskjermbildet, mens den andre metoden brukes til å legge til en ny knapp i brukergrensesnittet.

Det er et klart skille mellom metoder som er tilgjengelig utenfor en prosess og metoder som bare er tilgjengelig fra prosessen selv. Metodene som er synlige også utenfor en prosess må defineres i grensesnittet. Fordelen med en mobil agent er at den kan forflytte seg til en annen prosess og få tilgang til de interne metodene der. Dette er for eksempel tilfellet med de to nye metodene i `MorpFrame`.

I tillegg til å gjøre disse forandringene måtte grensesnittet til `AgendaList` klassen gjøres tilgjengelig utenfor sin egen prosess. Dette et dette som kalles å *remote enable* en klasse i Voyager (se også side 56).

5.2 Evaluering av prototypen

Prototypen har vist at det er mulig å utvide systemet dynamisk med nye funksjoner. Et problem er at det ikke er mulig å skifte ut funksjonalitet som allerede er i klienten. Prototypen støtter altså ikke alle typer vedlikehold. For eksempel hvis en av klassene inneholdt en Y2K feil, vill det det være vanskelig å bytte ut denne dynamisk.

Dette problemet skyldes måten Java linker sammen objektkode. Alle klasser som trengs i en Java applikasjon blir dynamisk lastet inn og linket av en *classloader* [Ven97, kapittel 8]. Hovedregelen er at programmerer ikke skal behøve å bry seg om lasting og linking av klasser i Java. Programmerer kan opprette nye objekter med nøkkelordet `new`, men det er ikke noen tilsvarende nøkkelord for å fjerne objekter. Frigjøring av minne ordnes automatisk av en teknikk som kalles *garbage collection*.

Et objekt kan ikke fjernes fra minnet før alle referanser til objektet er fjernet. Kjøretidssystemet i Java holder orden på alle referanser slik at objekter kan fjernes fra minnet når de ikke har referanser til seg. Først når alle objekter er fjernet kan klassedefinisjonen også fjernes.

En mulig løsning på problemet er å programmere seg et eget *classloader* objekt. Dette er det mulighet for i Java. Problemet er at man selv må holde orden på alle referanser til objekter man laster inn med *classloaderen*.

En annen svakhet ved MORP er at de utvidelsene agenten gjør ikke overlever termineringen av klienten. Når klienten avsluttes og startes opp igjen vil utvidelsen ikke lengre være tilgjengelig fordi den tidligere bare lå i minnet. I noen tilfeller er det praktisk å kun ha støtte for temporære utvidelser, men det bør også være mulig å gjøre utvidelsene permanente.

Dette problemet kan deles opp i to deler:

1. det å få lagret de nye klassene, og
2. det å sørge for at kjøretidssystemet oppdager klassene når klienten startes opp igjen.

Problem 1 kan løses ved å programmere et *classloader* objekt som lagrer de nye klassene den laster ned fra nettet. En annen teknikk er å overføre filene ved hjelp av FTP. I MORP får man her et nytt problem siden alle klassene er pakket inn i jar-filer. En jar-fil samler alle klassefilene i en fil. For å skifte ut en klasse må man lage en ny jar-fil. Dette blir normalt bare gjort en gang av utvikler. En jar-fil kan også ha et sertifikat. Dette sertifikatet forhindrer uvedkommende å redigere jar-filen.

Problem 2 kan løses ved at man skriver navnet på nye klasser til en fil. Ved oppstart må klienten lese filen og instansiere klassene. Dette er ikke implementert i MORP.

5.2.1 Ytelse

Tabell 5.1 viser de klassene som lastes over når agenten forflytter seg til en klient den ikke har vært på før. Selv om agenten i MORP kun har støtte for en enkel funksjon, bør det være et mål å minimere bruken av båndbredde. Dette gjelder spesielt hvis det er mange agenter som oppgraderer samtidig i et nettverk. Grunnen til dette er at mange agenter vil generere stor trafikk på nettet.

De to siste klassene (`MorpServer` og `MorpConsole`) i tabell 5.1 hører egentlig hjemme på tjenersiden i MORP. Det skal derfor ikke være nødvendig å laste disse klassene over til klienten. Allikevel bli de flyttet. Begrunnelsen for dette er at klassen `UpgradeAgent` importerer klassen `MorpServer` som igjen importerer klassen `MorpConsole`. Dette kunne ha vært unngått ved å programmere `UpgradeAgent` på en annen måte.

Navn	Størrelse i byte
morp/agent/UpgradeAgent.class	3785
morp/agent/IUpgradeAgent.class	253
morp/agent/PlaceException.class	390
morp/gui/AddAgendaButton.class	1445
morp/resource/tools/IAgendaList.class	321
com/objectspace/voyager/agent/ObjectAgent__Proxy.class	-
com/objectspace/voyager/agent/IObjectAgent.class	-
morp/agent/UpgradeAgent__Proxy.class	-
com/objectspace/voyager/Facets__Proxy.class	-
com/objectspace/voyager/IFacets.class	-
morp/gui/AddAgendaButton$AddAgendaButtonAction.class	977
morp/gui/AgendaDialog.class	3727
morp/resource/tools/AgendaList__Proxy.class	-
morp/resource/tools/Agenda.class	639
morp/MorpServer.class	2013
morp/gui/MorpConsole.class	2475
Samlet størrelse:	> 16025

Tabell 5.1: Tabellen inneholder navnet samt størrelsen på de klassene som blir lastet over nettverket når en agent forflyttes. De genererte proxy klassene var det ikke mulig å måle størrelsen på.

5.3 Oppsummering

Dette kapitlet har evaluert utviklingen av prototypen og selve prototypen. Evalueringen kan oppsummeres i disse observasjonene:

- Planlegging av vedlikeholdet skjer sent i utviklingsprosessen. Vedlikeholdbarheten ble ivaretatt ved å designe objektmodellen så generell som mulig slik at det skulle være mulig å legge til nye klasser.
- Subklassing en statisk teknikk som ikke egner seg for dynamiske endringer i et program. Et designmønster ble foreslått som løsning på dette.
- En annen observasjon gjelder grensesnittene i MORP. Her ble det forsøkt to forskjellige fremgangsmåter. Den første fremgangsmåten prøvde å gjøre grensesnittet og resten av applikasjonen uavhengig av hverandre. Dette førte til et enkelt grensesnitt, men klienten og tjeneren ble mer komplekse. I det andre forsøket ble klassedefinisjoner fra applikasjonen brukt direkte i grensesnittet. Dette førte til en enklere applikasjon og et enklere grensesnitt.
- For at den nye funksjonen (støtte for saksliste) skulle virke var det nødvendig å legge til et par nye metoder. Metodene kan ses på som

grensesnitt for nye funksjoner.

- En klar mangel ved prototypen er at den bare viser hvordan ny funksjonalitet kan legges til. Det å erstatte eksisterende funksjoner er ikke så lett. Dette skyldes teknikken Java bruker for å laste og linke klasser.
- Siste observasjon går ut på at størrelsen på agenten har nøye sammenheng med hvordan agenten er designet. Feil kan gjøre agenten større enn nødvendig. I systemer som bruker mange agenter kan dette ha konsekvenser for belastningen på nettverket. Det bør derfor være et mål å minimalisere størrelsen på agenten. Denne siste observasjonen peker kanskje også på at det er lurt å skille mellom selve agenten og koden som utgjør ny funksjonalitet. Dette er fordi koden vil utgjøre en vesentlig del av agenten.

Prototypen viser at mobile agenter kan brukes til adaptivitet fordi nye funksjoner kan legges til under kjøring av en applikasjon. I forhold til vedlikeholdstypene i kapittel 2 (se side 18) er det først og fremst perfektive endringer som støttes. Korrektive endringer er ikke støttet i MORP, men kunne har vært implementert ved å skifte ut klassene med feil i.

Neste kapittel introduserer en vedlikeholdsmodell som generaliserer konseptene fra MORP. Modellen tar blant annet også for seg hvordan konfigurasjonen (eller oppbygningen) til en applikasjon skal beskrives.

Kapittel 6

En vedlikeholdsmodell for mobile agenter

Prototypen i kapittel 4 er et eksempel på en applikasjon som bruker en mobil agent. En klasse (`UpgradeAgent`) representerte den mobile agenten, og oppgavene til agenten var programmert inn i klassen.

I dette kapitlet presenterer jeg en modell for hvordan vedlikehold kan utføres ved hjelp av mobile agenter. Modellen generaliserer idéene fra prototypen og forklarer ulike begreper og funksjoner. Hensikten er at denne vedlikeholdsmodellen skal kunne brukes generelt i distribuerte applikasjoner. Modellen er for eksempel uavhengig av Voyager som agentsystem eller Java som programmeringsspråk.

Det er flere mål med modellen:

- Modellen forklarer hvordan mobile agenter kan brukes til vedlikehold i distribuerte systemer.
- Modellen setter navn på begreper som brukes under vedlikehold. Dette gir også en oppdeling av de forskjellige rollene i et slik system. I modellen skilles det for eksempel mellom den delen av systemet som administrerer agentene, og de deler av systemet som blir vedlikeholdt.
- Modellen inneholder nye forslag som ikke ble utforsket i prototypen. Dette gjelder for eksempel hvordan konfigurasjonen til en applikasjon kan beskrives.

Hensikten med denne modellen er å gi et konseptuelt bilde av hvordan mobile agenter kan brukes til vedlikehold. Modellen kan derfor ikke brukes direkte i en konkret implementasjon. Den er for eksempel mer egnet til å beskrive hva som skjer under vedlikehold. Dette kan brukes under design og spesifikasjon av et program.

Modellen angir også en grafisk notasjon som kan ses på som et forslag til hvordan vedlikehold med mobile agenter kan visualiseres. Det er flere formålet med en grafisk notasjon:

- Å vise hvordan en eller flere agenter forholder seg til resten av applikasjonen. Dette gir et bilde av strukturen til et system som bruker mobile agenter. I MORP er det en enkel struktur som består av en tjener, en rekke klienter og en mobil agent.
- Å vise hvordan en agent kan endre eller oppgradere programvare. Dette gir et bilde av hendelsesforløpet til en agent. Agenten i MORP hadde et enkelt hendelsesforløp som gikk ut på å legge til klasser og migrere til neste klient.

6.1 Vedlikeholdsmodellen

Utgangspunktet for modellen er et distribuert system. Et distribuert system består av en samling autonome prosesseringselementer som ikke deler primærminne, men samarbeider ved å sende asynkrone meldinger over et nettverk slik det ble definert på side 36. En applikasjon som utnytter dette har jeg valgt å kalle en *distribuert applikasjon*.

Vedlikeholdsmodellen består av *noder*, *moduler* og *mobile agenter*:

- En node representerer en applikasjon som eksekverer på en maskin. Dette kan for eksempel være en klient eller en tjener i en distribuert applikasjon.
- Hver node er sammensatt av moduler. En modul representerer en objektfil, et bibliotek eller en annen del av inneholdet i en node.
- Mobile agenter forflytter seg mellom nodene i en distribuert applikasjon for å vedlikeholde nodene. Dette innebærer å skifte ut eller å legge til nye moduler.

Noder, moduler og mobile agenter har unike identifikatorer slik at det er mulig å skille dem fra hverandre. Hvordan identifiseringen utføres er opp til hver enkelt implementasjon. I MORP ble nodene identifisert ved hjelp av en IP-adresse og et portnummer.

6.1.1 Node

En node betegner den delen av en distribuert applikasjon som eksekverer på en maskin. I MORP er klienten et eksempel på en node. I tillegg til appli-

kasjonens primære funksjoner (for eksempel reservering av rom), inneholder hver node et agentsystem. Dette gjør det mulig å flytte mobile agenter mellom nodene.

Hver node har også forskjellige *grensesnitt*. Et grensesnitt beskriver de ulike tjeneste en node kan tilby andre noder. Det er kun gjennom grensesnittene at andre noder kan få tilgang til tjenester på en node. I [RWL95, side 173] blir grensesnittet mellom to noder beskrevet som *the surface area*. Dette området beskriver alt en konsument av en tjeneste trenger å vite fra tilbyderen av en tjeneste. Dette innebærer signaturer på funksjoner, synkronisering ved samtidig prosessering og restriksjoner i funksjonene. Et grensesnitt bør derfor spesifisere mer enn bare signaturen på en funksjon. I MORP tilbyr tjeneren et grensesnitt for å reservere rom, men spesifikasjonen av grensesnittet omfatter kun signaturen på metodene.

Foruten grensesnittene har en node *meta informasjon* som beskriver forskjellige sider ved noden og omgivelsene til noden:

Konfigurasjon: hver node består av et antall ulike moduler. En konfigurasjon er en oversikt over hvilke moduler en node består av. Konfigurasjonen kan også inneholde oversikt over hvilke versjoner av modulene som brukes.

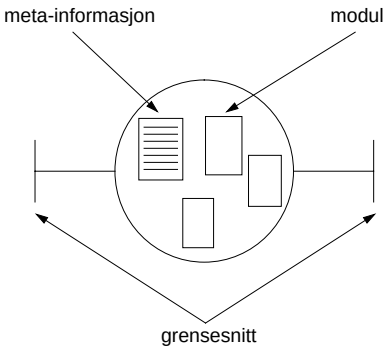
Lokale ressurser: noden bør ha oversikt over hvilke ressurser den har tilgjengelig. Dette kan være oversikt over lagringskapasitet, tilgjengelig minne, type prosessor, periferutstyr eller annen programvare. Denne informasjonen kan brukes av en agent til å tilpasse en oppgradering til maskin- og programvare på en node.

Felles ressurser: en node bør ha oversikt over hvilke felles ressurser den kan bruke. Dette kan være nåværende belastning i nettverket eller belastning på tjenere i systemet. Ut fra denne informasjonen kan en agent vente med å laste ned moduler til trafikken på nettet blir mindre.

Brukerpreferanser: en bruker kan spesifisere hva agenten får lov til å gjøre på en node.

Figur 6.1 viser en node med to grensesnitt, tre moduler og meta informasjon. Denne måten å tegne en node på er inspirert av RM-ODP (*Reference Model - Open Distributed Processing*) [ISO95] hvor objekter kan tegnes på denne måten. Dette kan være en måte å tegne en node, innholdet i noden og grensesnittene.

I prototypen i kapittel 4 ble ikke begrepene konfigurasjon, lokale ressurser, felles ressurser og brukerpreferanser brukt. Grunnen til dette var at alle nodene i MORP var såpass like at det ikke var nødvendig å beskrive for



Figur 6.1: Figuren viser en node med grensesnitt, moduler og meta informasjon.

eksempel konfigurasjonen for en node. I et større system med mange ulike programmer er meta informasjonen nødvendig for at agenten skal ha et sted å sjekke sammensetningen til noden eller omgivelsene rundt (for eksempel maskinvaren eller nettverket).

For å støtte adaptive applikasjoner er det dessuten nødvendig å ha oversikt over lokale og felles ressurser. For eksempel er det ikke hensiktsmessig for en agent å oppgradere en node med funksjonalitet for å spille av CD-ROM plater hvis maskinen ikke er tilkoblet en CD-ROM spiller.

En agent som for eksempel er på en håndholdt enhet (for eksempel en PDA) bør ha oversikt over nettverksforhold som båndbredde eller forsinkelse. Dette er viktig slik at agenten kan tilpasse applikasjonen på PDA'en til ulike nettverksforhold. Agenten vil for eksempel ikke laste ned store nye moduler hvis kommunikasjonslinken som brukes er en mobiltelefon med båndbredde på 9600 *bps*[1].

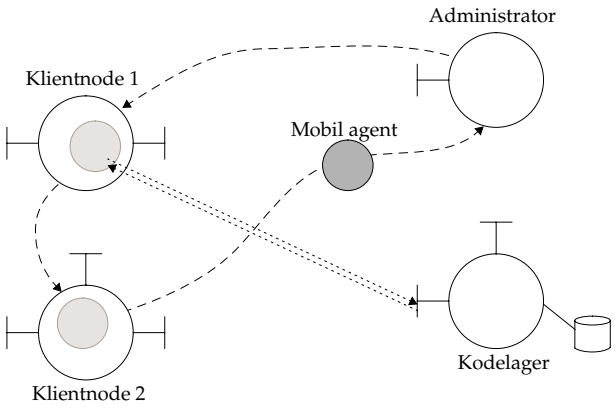
Videre skilller vedlikeholdsmodellen mellom fire typer noder:

Klientnode: er en node som skal eller blir vedlikeholdt av en agent. En mobil agent forflytter seg til en klientnode for å undersøke konfigurasjonen samt foreta oppgraderinger.

Administrator: er den noden som initialiserer og starter en agent. Agenten må settes opp med informasjon om hva den skal gjøre samt hvilke noder den skal til. Administrator er også ansvarlig for at vedlikeholdet blir utført.

Kodelager: her ligger nye moduler lagret. Det er fra denne noden at en agent kan få tak i nye moduler. Et kodelager fungerer som en database hvor utviklere kan registrere nye moduler.

[1] *bps* står for bits per sekund.



Figur 6.2: Figuren viser to noder som blir oppgradert (klientnodene). Administrator er den noden som sender ut en agent, mens et kodelager inneholder nye eller oppgraderte moduler. Den stiplede linjen viser hvordan agenten beveger seg fra node til node. Når agenten er i klientnode 1 gjøre den en forespørsel mot kodelageret for å laste ned en ny modul.

Tjenernode: dette er en node som tilbyr eksterne tjenester til agentene. Dette kan for eksempel være en navnetjener. Agenten kan bruke navnetjeneren til å finne adressen til en node.

Hensikten med å skille mellom disse fire typene noder er å dele opp funksjonaliteten slik at et system for vedlikehold kan bli lettere å implementere. Figur 6.2 viser et eksempel på en mobil agent som oppgraderer to noder. Figuren viser også en mulig fordeling av roller mellom nodene.

I MORP var det ingen klar oppdeling i rollene mellom nodene. Administratoren i MORP tilsvarer klassen `MorpServer` og de tilhørende klassene. `MorpServer` fungerte også som et kodelager for nye klasser som ble lastet ned av agenten.

I vedlikeholdsmodellen kan det være hensiktsmessig å fordele oppgavene over flere noder. I [RWL95, side 10] blir det påpekt at oppdeling[2] av en oppgave i flere underoppgaver kan føre til et mer håndterlig system, men at det også er en fare for at systemet blir for fragmentert. Oppdelingen av en node i forskjellige roller gjør at en programmerer kan konsentrere seg om en avgrenset del av oppgaven.

6.1.2 Modul

En modul er betegnelsen på en del av den interne strukturen i en node. Moduler kan være av forskjellige granularitet. Dette vil si at en funksjon

[2]Oppdeling av et problem i flere underproblemer blir ofte kalt *separation of concerns*.

på noen kodelinjer eller en klasse med flere hundre kodelinjer kan fungere som moduler. Eksempler på moduler kan være filer, komponenter eller klasser. Det som definerer en modul er at det må være en samhørighet i den interne strukturen til modulen. Dette kan være at de tilhører samme navnerom eller at de har relasjoner mellom seg. I Java vil en klasse eller en pakke være et eksempel på en modul. Oppdeling i moduler bestemmes i stor grad av den enkelte utvikler og baserer seg på erfaringer. Begrepet *modul* er inspirert fra en modell for komponentbasertutvikling [RWL95, side 173].

I modellen i dette kapitlet blir en modul sett på som en enhet. Igjen er dette en forenkling i forhold til hvordan det kan være i praksis. En modul kan for eksempel bestå av mange filer. Videre må modulene være i et ferdig kompilert format. Dette er fordi vi ønsker å dynamisk linke moduler inn i en kjørende applikasjon.

Moduler kan være relatert til hverandre på forskjellige måter [RWL95, 175]:

- To moduler kan ha et *peer-to-peer* forhold. Dette betyr at modulene bruker hverandre på en likeverdig måte.
- En modul kan bestå av flere mindre moduler som er på et lavere abstraksjonsnivå enn modulen de tilhører.
- En modul kan være en spesialisering av en annen mer generell modul.

En modul kan også ha det som jeg har valgt å kalle et *internt grensesnitt*. Et internt grensesnitt utgjør steder i koden hvor det er lagt til rette for utvidelser. Hensikten med et internt grensesnitt er å tilby en agent et grensesnitt hvor den kan skifte ut eller legge til moduler. Disse grensesnittene er bare tilgjengelig i det lokale adresserommet. Dette vil si at andre noder ikke har tilgang til de interne grensesnittene. Et eksempel på et internt grensesnitt kan være en metode for å legge til en knapp slik som beskrevet i MORP (se side 64).

Et kanskje bedre navn på interne grensesnitt er *plug-in points* slik det er beskrevet i [CLL99]. *Plug-in points* er steder i et klassehierarki hvor det er vanlig å utvide en applikasjon. Hensikten er å legge inn grensesnitt som støtter fremtidige utvidelser. Rent praktisk kan dette for eksempel gjøres ved å bruke `interface` nøkkelordet i Java.

I vedlikeholdsmodellen er inneholdet i en modul skjult. Det er kun gjennom grensesnittene til modulen at man kan få tilgang til det som skjer der. Denne egenskapen kalles innkapsling. Fordelen med innkapsling er at det kun er modulen selv som har kontroll på den interne strukturen.

I likhet med en node har en modul også meta informasjon. Denne informasjonen sier noen om hvilket versjonsnummer denne modulen har. Et versjonsnummer er nødvendig for at en agent skal kunne skille mellom forskjellige utgaver av den samme modulen. Alle moduler må dessuten ha et navn eller en identifikator som unikt skiller en modul fra alle andre. Meta informasjonen kan også omfatte avhengigheter denne modulen har. Dette gjelder både avhengigheter til andre moduler og til spesielle ressurser.

Et annet nyttig skille er *faste* og *utskiftbare* moduler. Alle moduler i en node kan ikke være utskiftbare fordi noen av modulene må danne *kjernen* i noden. Kjernen i en node er den minste samlingen av moduler en node må ha for å fungerer. I MORP må klassen `MorpClient` være en del av kjernen for klienten fordi den klassen inneholder `main` metoden som er nødvendig for å starte programmet.

I MORP er det ingen oppdeling i moduler utover klassebegrepet. En inndeling i forskjellige pakker ble gjort for å skille ulike deler av systemet fra hverandre, men agenten i MORP opererer ikke med pakker når den skal skifte ut eller legge til nye funksjoner. En modul i MORP tilsvarer en klasse.

6.1.3 Mobil agent

En mobil agent er begrepet for de programmene som har mulighet til å migrere mellom nodene. Dette er mulig fordi hver node har et agentsystem. Fordelen med å flytte en agent over til en annen node er at agenten og noden kan operere i samme adresserom. Dermed kan agenten få tilgang til de interne grensesnittene i de forskjellige modulene.

Oppgaven til en agent er å administrere og skifte ut moduler på nodene. Hvis vi holder oss til definisjonen på en mobil agent gitt i kapittel 3, må en agent være autonom, reaktiv og målrettet:

- Autonomitet vil si at agenten skal kunne fungere uten interaksjon fra brukeren. Dette kan man oppnå ved å utstyre agenten med regler for hva den skal gjøre og hvor den skal få sin informasjon fra.
- Reaktivitet vil si at agenten kan reagere på endringer i omgivelsene. Det mest aktuelle i forbindelse med vedlikehold er at agenten reagerer på endringer i moduler eller i kvaliteten på nettverksforbindelser.
- Målet til agentene i denne modellen er å vedlikeholde noder. Dette angir den pro-aktive egenskapen til agenten.

Mobile agenter kan flyttes eller forflytte seg selv mellom noder. Dette kalles *migrering* eller *reise* hvis man bruker begrepet slik det er beskrevet på

side 32. Før en agent starter på sin første migrering er den avhengig av følgende informasjon:

Reiserute: agenten må vite hvilke noder den skal besøke. Minimumskravet er at agenten vet adressen til den første noden.

Oppgavebeskrivelse: agenten må vite hvilke oppgaver den skal utføre på nodene. En agent kan være programmert til å gjøre en bestemt type jobb eller brukeren kan konfigurere dette selv.

Regler for reaktivitet: agenten må ha regler for hva den skal gjøre i forskjellige situasjoner. Eksempler på feil er for liten lagringskapasitet på noder, feil i moduler eller forsinkelse i nettverket.

Foruten dette inneholder agenten adressen til sin administrator samt adresser til eksterne tjenester. De eksterne tjenestene kan for eksempel brukes av agenten til å slå opp adresser eller til å koordinere sitt arbeid med andre agenter.

I MORP ble agenten satt opp med navnene til forskjellige noder. Adressene ble valgt ut via et grafisk brukergrensessnitt (se også figur 4.4 på side 51).

Beskrivelse av oppgaven til agenten var programmert i klassen til agenten (`UpgradeAgent`). I en mer fleksibel løsning er det ønskelig å konfigurere oppgavene til agenten mer enn i MORP. Dette gjelder også hva agenten skal gjøre ved forskjellige feilsituasjoner.

6.2 Funksjoner i vedlikeholdsmodellen

En agent har følgende faser:

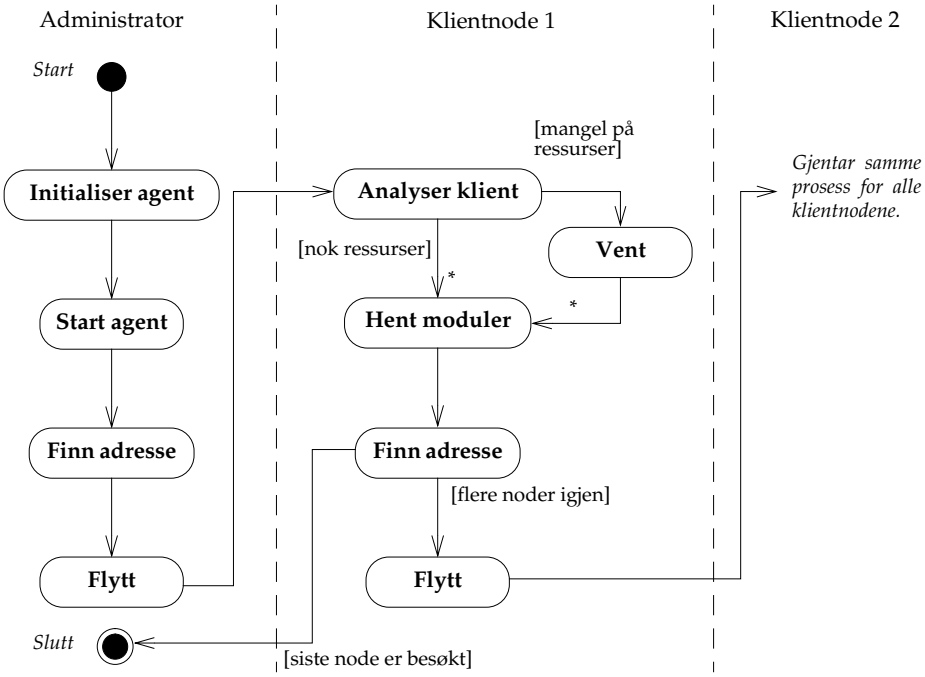
Analysefase: før en vedlikeholdsoppgave kan en agent utføre en analyse av de nodene den skal vedlikeholde. Agenten kan få oversikt over eksisterende moduler samt hvilke lokale ressurser som er tilgjengelig. Utfra denne informasjonen kan agenten tilpasse oppgraderingen til noden.

Distribusjonsfase: koden blir fraktet over nettverket. En agent kan velge å vente med nedlasting av kode hvis det er stor trafikk på nettet.

Installering: koden blir tatt i bruk. Dette innebærer å instansiere klasser og ta i bruk objektene i det eksisterende systemet.

Figur 6.3 viser et UML[3] aktivitetsdiagram [FS97] for en oppgradering. Agenten begynner på administratoren til venstre på figuren. Her blir den initia-

[3] UML står for *Unified Modeling Language*.



Figur 6.3: Aktivitetsdiagram som viser hvordan agenten opererer på forskjellige noder.

lisert med en reiserute og informasjon om hvilke oppgaver den skal utføre. Her kan man også legge inn regler for hva agenten skal ta hensyn til underveis (for eksempel at agenten ikke skal forflytte seg hvis forsinkelsen på nettverket overskrider en viss grense). Agenten starter så med å finne adresser for nodene. Dette er angitt via reiseruten til agenten.

De stiplede linjene viser hvor agenten skifter adresserom. Figuren viser en forenklet utgave av hvordan migrering mellom noder skjer. Det som virkelig skjer er at agenten blir serialisert og sendt over nettverket. Et agentsystem inneholder som regel metoder som skjuler en del av detaljene ved det å flytte en agent mellom adresserom. Dette gjør det enklere å programmere mobile agenter.

Når agenten har forflyttet seg over til en klientnode vil den foreta en analyse av hva som må gjøres på noden. Her kan agenten sjekke om nye moduler kan komme i konflikt med nodens konfigurasjon. En annen sjekk kan være om nedlasting av moduler skal skje nå eller senere. Dette avgjøres av reglene agenten fikk ved initialisering (regler for reaktivitet). Agenten kan legge igjen en kopi av seg selv som sørger for å foreta installering når det passer.

Agenten vil besøke alle nodene på sin reiserute. Når alle noder er besøkt kan agenten returnere til sin startnode eller den kan terminere seg selv på en av klientnodene.

6.2.1 Vedlikehold med flere agenter

Modellen i forrige avsnitt viste hvordan en agent kan utføre vedlikehold ved å migrere fra node til node. Ved å sende ut flere agenter i et nettverk kan oppdateringer gjøres samtidig. Dermed utnytter man de parallelle egenskapene til mobile agenter.

En annen måte å bruke flere agenter på er å fordele vedlikeholdsoppgavene på ulike agenter. En agent kan utføre analyse, en kan kontrollere distribusjon av moduler og en tredje agent kan installere moduler på en node. Denne oppgavefordelingen kan også gjøre det lettere å designe og programmere en agent. Istedenfor å ha en kompleks agent, kan oppgaven fordeles utover flere agenter. Agentene kan koordineres ved at de sender meldinger til hverandre eller at en administrator styrer hver enkelt agent.

6.3 Oppsummering

Dette kapitlet har presentert en vedlikeholdsmodell for mobile agenter. Modellen tar utgangspunkt i konseptene fra prototypen.

Sentrale begreper i modellen er noder, moduler og mobile agenter. En distribuert applikasjon består av flere noder spredt utover et nettverk. I MORP er for eksempel klienten en node.

Hver node består igjen av forskjellige moduler. Modulene er objektkode på hver node. Mobile agenter kan forflytte seg til noder for å skifte ut eller legge til nye moduler.

Formålet med modellen er å sette navn på de forskjellige delene som er relevante for vedlikehold i et distribuert system. Et mål er også å få fram konseptene ved å bruke mobile agenter til vedlikehold.

Kapittel 7

Diskusjon

I kapittel 4 ble det presentert en prototype for hvordan mobile agenter kan støtte utvidelse av en applikasjon under kjøring. Vedlikeholdsmodellen som ble presentert i kapittel 6 generaliserte konseptene fra prototypen. Dette kapitlet diskuterer forskjellige utfordringer med prototypen og vedlikeholdsmodellen.

En utfordring ved prototypen er å implementere støtte for forskjellig typer vedlikehold. For eksempel bør det være mulig å skifte ut noe av den eksisterende funksjonaliteten i MORP.

Prototypen er et lite eksempel og viser derfor ikke situasjonen i et distribuert system med mange noder. I et stort nettverk kan vedlikehold via agenter legge beslag på store deler av ressursene i et nett. Dette kapitlet diskuterer konsekvenser for ressursfordeling, konsistens, åpenhet og skalerbarhet i distribuerte systemer som bruker mobile agenter til vedlikehold.

Arkitekturen til et system er avgjørende for vedlikeholdbarheten. Prototypen viste at en generell modell er lettere å utvide og tilpasse. Samtidig bør ikke en modell for vedlikehold dominere arkitekturen. Dette kan få konsekvenser for ytelsen til systemet.

7.1 Ulike typer vedlikehold i distribuerte systemer

Fra kapittel 2 har vi fire typer endringer: korrektive, adaptive, perfektive og preventive. Teoretisk er det interessant å skille mellom disse typene fordi de sier noe om hva vi legger i begrepet vedlikehold. Det som også skiller disse typene er omfanget av endringene. For eksempel kan korrektive endringer være små kodefeil, mens perfektive endringer kan være større utvidelser. I praksis vil en vedlikeholdsjobb omfatte en kombinasjon av disse endringene.

Det vi tradisjonelt forbinder med vedlikehold, er endring etter at systemet

er tatt i bruk av kunden. Dette innebærer blant annet å utvikle, distribuere og installere endringer. Denne oppgaven har sett nærmere på hvordan distribusjon og installering kan automatiseres ved å bruke mobile agenter. En vedlikeholdsprosess med mobile agenter kan se slik ut:

- En forespørsel om en ny funksjon eller retting av en feil blir registrert av utviklerne. Denne endringen blir kanskje implementert i form av en eller flere moduler. Disse modulene blir så testet i et testmiljø. Når modulene har tilfredsstillende kvalitet, blir de gjort tilgjengelig i kodelageret (se side 72).
- En eller flere mobile agenter blir satt i gang for å oppgradere noder med de nye modulene. Dette kan innebære både en analysefase og en distribusjonsfase slik det ble diskutert på side 76.
- For å installere modulene kan agenten kalle et internt grensesnitt som gjør at de nye modulene blir tatt i bruk.

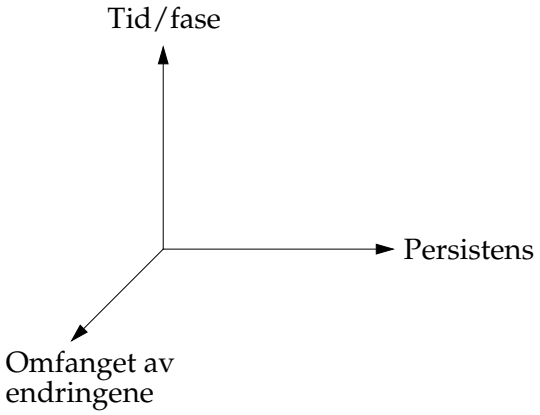
Denne oppgaven presenterer en teknikk for å distribuere og installere moduler. Nedlasting av moduler kan skje over nettverket ved å bruke vanlige kommunikasjonsprotokoller. For at en node skal ta i bruk nye moduler må noden tilby et passende grensesnitt for agenten. Dette ble kalt interne grensesnitt i vedlikeholdsmodellen (se også side 74).

Det er også viktig å være klar over hva slags vedlikehold en applikasjon skal støtte. I MORP ble modulene tatt i bruk under eksekvering, men modulene ble ikke lagret for senere bruk. Ut fra erfaringene med MORP og vedlikeholdsmodellen har jeg funnet fram til tre kontrollspørsmål som kan brukes til å finne ut av hva slags vedlikehold en applikasjon skal støtte:

1. Skal endringene skje i kjøretid eller må systemet startes på nytt for at endringene skal ha noen effekt? Dette kaller jeg for henholdsvis *dynamisk* og *statisk* endring.
2. Skal endringene lagres slik at de også er tilgjengelig når systemet startes på nytt? En endring kan være enten *permanent* eller *ikke-permanent*.
3. Innebærer endringene kun utvidelser eller også forandring av det eksisterende systemet? Det er mulig å kategorisere endringer ut fra omfanget.

Disse tre faktorene er også vist på figur 7.1. Aksen merket *tid/fase* illustrerer om endringen skal skje under kjøring eller ikke, mens aksen merket *persistens* illustrerer om endringen er permanent eller ikke. Aksen merket *omfang av endringene* illustrerer hvor stor del av systemet som skal være tilgjengelig for vedlikehold. Forskjellige kombinasjoner av disse faktorene bestemmer

egenskapene til blant annet agentsystemet, kjøretidssystemet og programmeringsspråket. Valg av type vedlikehold påvirker også arkitekturen på systemet.



Figur 7.1: Forskjellige faktorer som bestemmer kompleksiteten til en endring.

Statisk endring: betyr at systemet endres når det ikke kjører. Dette vil si at systemet betraktes utenfra som en mengde filer. For at endringene skal ha noen effekt må systemet startes på nytt.

Dynamisk endring: skjer mens systemet eksekverer. Klasser og biblioteker blir linket dynamisk. Endringene skjer derfor uten at man trenger å starte applikasjonen på nytt. Å legge til en funksjon for sakslister i MORP er et eksempel på dynamisk vedlikehold. Dynamisk vedlikehold krever at kjøretidssystemet er i stand til å linke klasser dynamisk. Applikasjonen må også vite hvordan den skal ta i bruk nye klasser eller moduler.

Permanent endring: vil si at nye funksjoner blir tatt vare på slik at de kan brukes ved senere kjøringer av systemet.

Ikke-permanent endring: betyr at nye moduler ikke blir lagret, men kun ligger i minnet. Når applikasjonen terminerer vil modulene også bli slettet fra minnet. I MORP vil for eksempel funksjonaliteten for å legge inn en saksliste på hvert møte forsvinne når klienten terminerer.

Omfanget av endringene: kan betegnes ut fra hvor stor del av det eksisterende systemet som endres. Spesielt kan man se på om det er nødvendig å skifte ut hele applikasjonen (det vil si at nesten alle modulene er utskiftbare). I vedlikeholdsmodellen ble det foreslått at noen av modulene må danne kjernen i applikasjonen (se også side 75).

Skillet mellom statisk og dynamiske endringer er interessant av flere grunner. Hvis endringene utføres dynamisk kan dette bety at man slipper å ta ned et distribuert system for at endringene skal ha noen effekt. Dynamisk vedlikehold åpner også for *adapsjon*. Adapsjon vil si at deler av et system kan endre funksjonalitet etter hvert som omgivelsene forandres.

Ulempen med dynamisk endring er at systemet må tilby et grensesnitt spesielt for endringer. I vedlikeholdsmodellen ble dette betegnet som interne grensesnitt fordi de ikke behøver å være tilgjengelig utenfor adresserommet til noden. Et eksempel på dette kan være et metode som legger til en ny knapp eller et ekstra vindu.

En permanent endring vil typisk være en endring for å rette en feil eller legge til en ny funksjon. En ikke-permanent endring kan være nyttig i tilfeller hvor ny funksjonalitet kun skal være tilgjengelig i en kort periode. Under planlegging av et system er det viktig å ta hensyn til at stadige nedlastinger av moduler kan belaste nettverket unødig mye.

I vedlikeholdsmodellen ble det skilt mellom faste og utskiftbare moduler (se side 75). Faste moduler betegner deler av koden som ikke kan skiftes ut under kjøring. Et eksempel på dette er de klassene i MORP som blir lastet inn ved oppstart. Disse klassene er det ikke mulig å erstatte under kjøring på grunn av kjøretidssystemet i Java.

Ved utvikling av et system kan det være vanskelig å gjette seg fram til hvilke deler av systemet som engang må endres. Det kan derfor være vanskelig å implementere interne grensesnitt i alle moduler. En statisk endring kan derimot skifte ut alle deler av et system fordi systemet betraktes som en mengde filer.

7.2 Utfordringer ved vedlikeholdsmodellen

Vedlikeholdsmodellen i kapittel 6 beskrev en rekke konsepter ved det å bruke agenter til vedlikehold. Modellen er en abstrakt beskrivelse, og den tar derfor ikke for seg teknologien for å støtte begreper i modellen.

Forskjellige utfordringer jeg har funnet i vedlikeholdsmodellen er:

- Hvordan skal man instruere en mobil agent om hva den skal gjøre? Agenten må settes opp med regler for oppgaven den skal gjøre samt hvordan den skal reagere på endringer i omgivelsene.

- Hvordan beskrives meta informasjonen til moduler og noder? Meta informasjonen omfatter både konfigurasjon, ressurser og brukerpreferanser.

- Hvordan får man oversikt over lokale og felles ressurser? Noen ressurser er lette å måle (for eksempel ledig lagringsplass), mens andre ressurser, som for eksempel forsinkelse, er vanskeligere å måle.

- Hvordan kan man sikre konsistens mellom noder? I en distribuert applikasjon kan en node være avhengig av en beregning fra en annen node. Under vedlikehold er det derfor viktig å sikre seg mot at nodene gjør gale beregninger.

- Hva er konsekvensene for arkitekturen til en node? Det å ta i bruk et agentsystem får konsekvenser for hvordan applikasjonen er konstruert.

Disse utfordringene blir diskutert videre i dette kapitlet.

7.2.1 Instruksjon av en mobil agent

En utfordring i MORP er å spesifisere hva agenten skal gjøre og hvordan den skal reagere på forskjellige situasjoner (det vil si at agenten er reaktiv). Hovedoppgaven til agenten er å skifte ut moduler. For å gjøre dette må agenten ha tilgang til meta informasjonen for den nye modulen og for noden den skal vedlikeholde. I utgangspunktet er det to måter å instruere en agent på [LD98]:

1. å ha agenter som er spesialskrevet for forskjellige oppgaver, eller[1]

2. å ha generelle agenter som kan konfigureres til å gjøre en spesiell oppgave[2].

Begge typer agenter har sine fordeler og ulemper. En generell agent kan brukes til mange vedlikeholdsoppgaver, men den utfører kanskje oppgavene mindre effektivt enn en spesial skrevet agent.

I [LD98] blir det foreslått at all konfigurering av agenter kan gjøres gjennom en nettleser. For eksempel kan agenten presentere et skjema[3] hvor en bruker kan velge hva en agent skal gjøre. Fordelen ved å bruke en nettleser er at den bruker et kjent brukergrensesnitt samtidig som nettlesere finnes for de fleste plattformer.

[1] Dette kalles *special-purpose agents* i [LD98].
[2] Dette kalles *general-purpose agents* i [LD98].
[3] Et skjema i HTML kalles gjerne for *forms*.

7.2.2 Beskrivelse av meta informasjon

I vedlikeholdsmodellen består en node av forskjellige moduler. Meta informasjonen beskriver hvilke moduler og versjoner en node består av. En modul har også meta informasjon som blant annet beskriver avhengigheter til andre moduler.

Meta informasjonen bør kunne leses av en agent både ved statiske og dynamiske endringer. Applikasjonen og dens meta informasjon bør derfor være adskilt.

En måte å uttrykke en konfigurasjon på er å bruke OSD (*Open Software Description*) [vHPT97]. OSD er et forslag til standard for hvordan man kan beskrive konfigurasjoner for applikasjoner. En konfigurasjon blir uttrykt ved hjelp av en OSD fil. En OSD fil er en tekstfil skrevet i henhold til XML (*eXtensible Markup Language*) [BPSM98]. Fordelen ved å bruke XML er at man selv kan legge til ekstra felter i OSD filen. Eksempler på felter i en OSD fil er `OSVERSION` og `PROCESSOR`. Disse beskriver krav til henholdsvis operativsystem og prosessor for en konfigurasjon.

En OSD fil kan også beskrive avhengigheter til andre pakker. Dette er nyttig hvis en pakke er avhengig av et annet program eller bibliotek for å fungere. Dette gjøres ved å referere til en annen OSD fil.

En tilsvarende løsning kan brukes for å beskrive meta informasjonen for moduler. Et krav bør være at denne informasjonen er lett å finne fram i samtidig som den bør være lett å forandre for en agent.

7.2.3 Oversikt over ressurser

For at vedlikehold ved hjelp av mobile agenter skal være effektivt, er det nødvendig for agenten å ha oversikt over hvilke ressurser den har til rådighet. I vedlikeholdsmodellen ble det skilt mellom lokale ressurser og felles ressurser. Lokal ressurser er for eksempel ledig plass på harddisken, mens felles ressurser kan være båndbredde eller forsinkelse.

En mobil agent bruker denne informasjonen til å bestemme hvordan og når den skal utføre vedlikehold. Agenten kan for eksempel velge å vente med å utføre vedlikeholdet til et tidspunkt hvor nettverkstrafikken er mindre.

I [ARS97] er det et forslag til hvordan ressurser kan monitoreres (KOMODO). Dette forslaget går ut på at hver node i et distribuert system har en ekstra prosess som kalles for en monitor. Denne monitoren kan i prinsippet måle både lokale og felles ressurser. I prototypen som er beskrevet i [ARS97] er det bare måling av forsinkelse som er implementert.

Måten KOMODO fungerer på er at agenter abonnerer på informasjon om

forskjellige ressurser. Denne forespørselen gjøres til den lokale monitoren. Hvis den lokale monitoren ikke klarer å håndtere forespørselen, blir en av de andre monitorene kontaktet. Et eksempel på dette er måling av forsinkelse mellom to noder.

Videre er det to typer monitorering:

1. *on-demand*, eller
2. kontinuerlig monitorering.

Den første typen gir bare melding om status for en ressurs, mens den andre typen monitorering gir en strøm av informasjon om den ønskede ressurs. *On-demand* monitorering kan for eksempel brukes til å sjekke om det er ledig lagringskapasitet på en node. Kontinuerlig monitorering kan for eksempel brukes til å overvåke en nettverksforbindelse.

Et viktig prinsipp i KOMODO er at monitoreringen ikke skal gå utover annen prosessering på noden. For å begrense ressursbruken er det innført en ordning om at agenter må fornye abonnementet med jevne mellomrom. Det er også implementert en grense på hvor mange monitoreringer som kan foregå samtidig.

Ulempen med KOMODO er at man må ha to prosesser på hver node. En løsning på dette hadde vært å legge monitoreringsfunksjonen sammen med agentsystemet.

7.2.4 Konsistens mellom noder

Et problem ved vedlikehold i distribuerte systemer er inkonsistens. Dette skjer for eksempel hvis en node skifter ut en kommunikasjonsprotokoll. Man kan tenke seg et scenario hvor en mobil agent oppgraderer en protokoll på node A, men ikke på node B. Da kan nodene A og B få problemer med å kommunisere hvis protokollene ikke er kompatible.

Dette problemet gjelder ikke bare kommunikasjonsprotokoller. Et distribuert system kan under oppgradering være ustabilt fordi forskjellige noder oppfører seg forskjellig i forhold til hvilken versjon av programvaren de kjører. Dette kan føre til at resultatet av beregninger blir feil.

Når man bruker mobile agenter til å utføre oppgradering blir det viktig å teste modulene som skal spres ved hjelp av mobile agenter. Under spredning av moduler kan den mobile agenten programmeres til å undersøke den gjeldende konfigurasjonen på noden den befinner seg på. Agenten kan også få ansvaret for å oppgradere slik at versjoner ikke kommer i konflikt med hverandre lokalt på den noden agenten befinner seg.

For å opprettholde konsistens mellom nodene er det viktig å koordinere vedlikeholdet. Dette kan gjøres av en ekstern prosess som styrer agentene eller de kan koordinere seg selv. JADA [CR97] et et forslag til hvordan agenter kan koordineres i Java. Via det som kalles for et *objectspace* kan tilgangen til objekter synkroniseres mellom prosesser i et distribuert system.

Et annet problem med å oppgradere applikasjoner er at man bør være kompatibel med eldre versjoner av systemet. Det kan være alt fra å forstå eldre filformater til å støtte programmer skrevet for en annen plattform. Noen ganger krever man at programvaren skal være bakover kompatibel.

7.2.5 Konsekvenser for arkitekturen

Erfaringene fra prototypen har vist at det er nødvendig med interne grensesnitt for å støtte dynamisk utvidelse av en applikasjon. I MORP var metoden `addButton` et eksempel på et internt grensesnitt.

Før en agent kan bruke et internt grensesnitt i en modul, er det nødvendig å få tak i en referanse til den aktuelle modulen. I MORP ble dette programmert som metoden `getInstance`.

En lignende løsning blir presentert i et rammeverk for dynamisk komponent oppgradering (DCUP - *Dynamic Component Updating*) [FP97]. Denne modellen ser på en applikasjon som et tre av komponenter. En komponent er delt inn i en permanent og en utskiftbar del. Dette er for å skille mellom kontrolldelen (permanent) og den funksjonelle delen (utskiftbar) av en komponent. Kontrolldelen i en komponent administrerer de utskiftbare delene.

Videre må alle utskiftbare deler i DCUP implementere grensesnittet `Reachable`. Dette er nødvendig for å få tak i referanser til de ulike objektene.

Ulempen ved å følge et rammeverk som DCUP er at alle komponenter må følge reglene for hvilke grensesnitt de skal implementere. Alle utviklere må derfor lære seg et spesielt rammeverk og følge dette.

I prinsippet kan man programmere en applikasjon som kun består av metoder for å håndtere linking og utskifting av moduler. Applikasjonen har da ingen funksjonalitet i utgangspunktet, men kan laste ned moduler etter hvert. Ulempen med en så generell applikasjon er at det kan ta lang tid å starte applikasjonen i tillegg til at den kan bli stor.

Et bedre alternativ er kanskje å implementere grunnleggende funksjonalitet i applikasjonen. I MORP er for eksempel romreservering en grunnleggende funksjon. Ekstra funksjoner kan så legges til dynamisk. Dette kan gjøre applikasjonen mer robust ovenfor feil i en dynamisk oppdatering. Selv om de nye modulene ikke virker har man fortsatt tilgang til den grunnleg-

gende funksjonaliteten.

7.3 Refleksjoner rundt bruk av mobile agenter til vedlikehold

I avsnitt 1.1.1 ble forskjellige teknikker for vedlikehold presentert. Teknikkene varierer fra manuell til automatisk nedlasting og installering av programvare:

- Den enkleste formen for å spre kode kan gjøres via applikasjoner som støtter FTP. En bruker kan laste ned programvare og installere denne. For eksempel kan denne teknikken brukes for å laste ned nye utgaver av et program. Ulempen er at brukeren selv må installere programvaren og sørge for at annen nødvendig programvare også er installert.
- En annen teknikk som brukes spesielt i spill og operativsystemer er *patching*. Dette er en teknikk for å rette feil eller legge til midlertidige endringer i et program. Igjen er det brukeren som har ansvaret for å starte en *patch* samt sørge for at annen nødvendig programvare er installert.
- Castanet fra Marimba kan brukes til å installere programvare i et nettverk. Uavhengig av programvaren kan Castanet brukes til å oppgradere og administrere programvare i et nettverk. Teknikken støtter ikke dynamisk endring av en applikasjon.
- Det finnes også mange teknikker for å oppgradere spesifikke applikasjoner. Eksempler på dette er *plug-ins* fra Netscape og *AutoUpdate* fra RealPlayer G2.

Teknikken med å bruke mobile agenter til vedlikehold slik det er presentert i denne oppgaven tar hensyn til endringer i programvare helt fra starten av et utviklingsprosjekt. Ved å bruke vedlikeholdsmodellen i kapittel 6 samt et agentsystem kan det bli lettere å endre en applikasjon senere.

Et agentsystem er en abstraksjon over kommunikasjonsprotokollene som gjør det lettere å flytte kode og tilstand mellom noder. Vedlikeholdsmodellen gjør det lettere å planlegge hvordan vedlikeholdet skal skje.

Et agentsystem kan brukes som basis for en distribuert applikasjon. I applikasjonen kan mobile agenter også brukes til andre oppgaver enn vedlikehold. Forskjellige bruksområder ble presentert på side 29. For eksempel kan mobile agenter både brukes til vedlikehold og til å samle inn informasjon i et nettverk.

Mobile agenter er også egnet til å støtte adaptive applikasjoner. Per i dag finnes det ikke mange applikasjoner som trenger slik adaptivitet, men økt bruk av håndholdte enheter som mobiltelefoner og PDA'er tilsier at dette kan komme. Adaptivitet er viktig når tjenstetilbudet må variere for eksempel på grunn av skiftende nettverksforhold.

7.4 Oppsummering

Dette kapitlet har diskutert forskjellige utfordringer ved å bruke mobile agenter til vedlikehold. Utfordringene varierer i forhold til hva slags type vedlikehold som skal støttes.

Vi har sett at vedlikeholdsbegrepet slik vi kjenner det i dag ikke er tilpasset nye applikasjoner. Dette gjelder for eksempel det å støtte adaptivitet i applikasjoner. Vedlikehold via mobile agenter dreier seg ikke bare om å distribuere og installere ny kode, men også om å klargjøre systemer for fremtidig vedlikehold.

Tre spørsmål ble introdusert for å klargjøre hva slags vedlikehold en applikasjon skal støtte. Disse spørsmålene gikk ut på å klargjøre når endringene skjer (enten dynamisk eller statisk), om vedlikeholdet skal være permanent og omfanget av endringene.

Kapitlet så også på ulike utfordringer ved vedlikeholdsmodellen:

- Hvordan skal man instruere en agent? En måte er å instruere agenten via en nettleser.
- Hvordan skal meta informasjonen beskrives? OSD filer som bruker XML kan brukes til dette.
- Hvordan får man oversikt over lokale og felles ressurser? Et forslag er å bruke en ekstern monitor. Agenter som ønsker oversikt over en eller flere ressurser må abonnere på tjenester fra monitoren.
- Hvordan kan man sikre konsistens mellom noder? En løsning er å bruke et koordineringssystem for å synkronisere objekter i et distribuert system.
- Hva er konsekvensene for arkitekturen? Arkitekturen må følge visse regler for å støtte dynamisk endring. Konsekvensene for arkitekturen er derfor avhengig av hvor dynamisk applikasjonen skal være. For å dynamisk legge til en ny funksjon er det nødvendig med metoder for å få tak i referanser til moduler som har interne grensesnitt.

Kapittel 8

Konklusjon

Målet for denne oppgaven var å undersøke hvordan mobile agenter egner seg til vedlikehold i distribuerte systemer.

Vedlikehold blir ofte forbundet med feilretting, men opptil 50 prosent av vedlikeholdet er det som kalles perfektive endringer. Perfektive endringer går ut på å legge til funksjonalitet slik at systemet blir bedre å bruke. Videre er det slik at et system utvikler seg via evolusjon og ikke-planlagte utvidelser. Det er derfor vanskelig å planlegge vedlikehold. Systemutviklingsmetoder inneholder også lite støtte for vedlikehold. Teknikker for å øke vedlikeholdbarheten tar ofte utgangspunkt i forskjellige former for gjenbruk. Gjenbruk går ut på å bruke analyse, design og kode fra andre lignende applikasjoner.

En mobil agent ble definert som en del av et program som er uavhengig av stedet den ble instansiert. Mobile agenter er egnet til vedlikehold fordi en agent er autonom, reaktiv og målrettet. Den kan derfor forflytte seg mellom noder i et distribuert system for å oppgradere. Rent teknisk er ikke en agent forskjellig fra vanlige teknikker for datakommunikasjon. Forskjellen ligger i hvordan vi organiserer kommunikasjonen. Fordelen med agenter er at de kan utnytte lokal kommunikasjon og få tilgang til den interne strukturen i en node.

Prototypen i kapittel 4 var et eksempel på en distribuert applikasjon som brukte mobile agenter til å utvide funksjonaliteten. Prototypen ga praktiske erfaringer med mobile agenter og vedlikehold. Et av problemene var å skifte ut funksjoner i prototypen. Evalueringen av prototypen viste at det var nødvendig å legge til nye metoder for å støtte dynamisk endring av applikasjonen.

I kapittel 6 ble det presentert en vedlikeholdsmodell. Modellen viser konseptene for vedlikehold med mobile agenter. Viktige begreper var noder, moduler og mobile agenter. Modellen pekte også på noen utfordringer. Dette gjelder for eksempel hvordan agenter skal instrueres eller hvordan

oppbygningen til en applikasjon skal beskrives. Disse utfordringene ble diskutert i kapittel 7.

8.1 Hva var målene for oppgaven?

Målet for denne oppgaven var å undersøke hvordan mobile agenter egner seg til vedlikehold. Denne problemstillingen ble delt opp i disse delproblemene:

- *Hvordan kan mobile agenter støtte vedlikehold i et distribuert system?*

  Mobile agenter kan forflytte seg mellom forskjellige adresserom. En agent kan derfor få tilgang til objekter i en applikasjon. Agenten kan skifte ut disse objektene hvis kjøretidssystemet støtter dynamisk linking av klasser. For at dette skal være mulig må også applikasjonen inneholde metoder for å legge til eller skifte ut funksjoner.

- *Hvordan kan mobile agenter støtte adaptive applikasjoner?*

  Mobile agenter kan støtte både dynamisk og statisk endring av en applikasjon. Adapasjon vil si at et program tilpasser seg omgivelsene. Agenter kan overvåke omgivelsene og tilpasse en applikasjon ved å laste ned protokoller eller funksjoner. For at dette skal være mulig må arkitekturen være slik at den kan ta i bruk ny funksjonalitet. En utfordring er å designe nye moduler og agenter slik at de bruker minst mulig båndbredde.

- *Hvordan kan mobile agenter gjøre applikasjoner mer åpne for endringer?*

  Et agentsystem kan brukes i arkitekturen til et system for å gjøre det mer åpent for endringer. Vedlikeholdsmodellen som ble presentert i kapittel 6 kan brukes i en systemutviklingsmetode for å planlegge vedlikehold. Videre kan kontrollspørsmålene fra kapittel 7 brukes til å klargjøre hva slags vedlikehold applikasjonen skal støtte.

8.2 Videre arbeid

Denne oppgaven har sett på mange områder som kan utforskes videre. Forslag til videre arbeid er:

- Hvordan kan vedlikehold ved hjelp av mobile agenter kombineres med komponenter? En komponent tilsvarer det som kalles for modul

i vedlikeholdsmodellen. En idé er å lage et system hvor man kan kjøpe eller leie komponenter. Når man kjøper en komponent kan denne betraktes som en permanent endring. Leie av en komponent kan betraktes som en ikke-permanent dynamisk endring. Dette er det mulig å knytte opp mot elektronisk handel.

- Denne oppgaven har ikke sett nærmere på sikkerhet i forbindelse med vedlikehold. Sikkerhet er viktig for at folk skal få tillit til et system og mobile agenter generelt. Dynamisk endring av en applikasjon kan misbrukes på mange måter. Et problem er hvordan man skal unngå at nye moduler inneholder virus.

- Et annet forslag til videre arbeid er å kombinere distribuerte objekter og objektorienterte databaser. Fordelen med dette er at objekter kan lagres persistent. I forbindelse med vedlikehold kan en objektorientert database brukes til å lagre moduler samt agenter. Et mål må være at det å lagre og finne fram objekter er transparent for programmerer.

- Denne oppgaven har ikke sett spesielt nøye på det å utnytte flere mobile agenter i parallell. Samtidig kjøring krever at agenten koordineres. Et formelt språk kan brukes til å spesifisere hvordan flere agenter skal samvirke.

- For å gjøre utviklingen av mobile agenter lettere kan man utnytte et CASE[1] verktøy. Dette kan også kombineres med gjenbrukbare komponenter. Et CASE verktøy kan også visualisere en vedlikeholdsoppgave.

- Ved bruk av mobile agenter til vedlikehold, slik som beskrevet i denne oppgaven, er det viktig å verifisere at vedlikeholdet blir utført korrekt. Dette kan være vanskelig å teste fordi nodene i et system kan ha forskjellige konfigurasjoner. Et område for videre forskning kan derfor være å finne fram til metoder for å teste moduler slik at dette blir tatt hensyn til.

[1] CASE står for *Computer Aided Software Engineering*.

Bibliografi

[ARS97] Anurag Acharya, M. Ranganathan og Joel Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. I Vitek og Tschudin [VT97], side 111–130.

[Ber96] Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, februar 1996.

[Boe86] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):22–32, august 1986.

[BPSM98] Tim Bray, Jean Paoli og C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Recommendation, The World Wide Web Consortium, 1998. `http://www.w3.org/TR/REC-xml` (27.september 1999).

[BRJ99] Grady Booch, James Rumbaugh og Ivar Jacobsen. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[BS98] Gordon Blair og Jean-Bernard Stefani. *Open Distributed Processing and Multimedia*. Addison Wesley, 1998.

[CDK94] George Coulouris, Jean Dollimore og Tim Kindberg. *Distributed Systems — Concepts and Design*. Addison Wesley, 2. utgave, 1994.

[CHK97] David Chess, Colin Harrison og Aaron Kershenbaum. Mobile agents: Are they a good idea? I Vitek og Tschudin [VT97], side 25–45.

[CLL99] Peter Coad, Eric Lefebvre og Jeff De Luca. *Java Modeling in Color With UML : Enterprise Components and Process*. Prentice Hall, 1999.

[CR97] Paolo Ciancarini og David Rossi. Jada: Coordination and Communication for Java Agents. I Vitek og Tschudin [VT97], side 213–226.

[Cro96] Jon Crowcroft. *Open Distributed Systems*. UCL Press, 1996.

[DM93] Bo Dahlbom og Lars Mathiassen. *Computers in Context*. NCC Blackwell, 1993.

[Dol97] Jean Dollimore. Object-based Distributed Systems CORBA, ORBIX and Java RMI. Draft material for 3rd edition of Distributed Systems — Concepts and Design http://www.dcs.qmw.ac.uk/research/distrib/dsbook (1. november 1999), 1997.

[Fla97] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, 2. utgave, 1997.

[Fow96] Martin Fowler. *Analysis Patterns : Reusable Object Models*. Addison Wesley, oktober 1996.

[FP97] Radovan Janecek Frantisek Plasil Dusan Balek. DCUP: Dynamic Component Updating in Java/CORBA Environment. Rapport, Dep. of SW Engineering, Charles University, Prague, 1997.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco og Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on software engineering*, 24, 1998.

[FS97] Martin Fowler og Kendall Scott. *UML Distilled: Applying the Standard Modeling Object Language*. Object Technology Series. Addison Wesley, 1997.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson og John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, 1996.

[GHN+97] Shaw Green, Leon Hurst, Brenda Nangle, Dr. Pádraig Cunnigham, Fergal Somers og Dr. Richard Evans. Software Agents: A review, mai 1997. http://www.cs.tcd.ie/research_groups/aig/iag/pubreview.ps.gz (21. oktober 1999).

[GJS96] James Gosling, Bill Joy og Guy L. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[Int97] IEEE Internet Computing, juli - august 1997.

[ISO95] ISO/IEC. Open Distributed Processing - Reference Model - Part 1: Overview. Rapport, ISO, 1995. ftp://ftp.dstc.edu.au/pub/arch/RM-ODP/PDFdocs/part1.pdf (7. oktober 1999).

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, oktober 1997.

[Jr.87] Frederick P. Brooks Jr. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, april 1987.

[JSLJ99] Bill Janssen, Mike Spreitzer, Dan Larner og Chris Jacobi. *ILU 2.0 beta1 Reference Manual*. Xerox, 1999. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html` (21. oktober 1999).

[KGN+97] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla og George Cybenko. AGENT TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, side 58–67, juli - august 1997.

[KJ98] Michael Knapik og Jay Johnson. *Developing Intellingent Agents for Distributed Systems*. McGraw-Hill, 1998.

[Lan98] Danny. B. Lange. Mobile Objects and Mobile Agents: The Future of Distributed Computing? *Lecture Notes in Computer Science*, 1445:1–12, 1998.

[LB85] M. M. Lehman og L. A. Belady. *Program evolution: processes of software change*. Academic Press, London, UK, 1985.

[LD98] Anselm Lingnau og Oswald Drobnik. Agent-User Communications: Requests, Results, Interaction. I Rothermel og Hohl [RH98], side 209–221.

[LO98] Danny B. Lange og Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[MBB+98] Dejan Milojičić, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sankar Virdhagriswaran og Jim White. MASIF - The OMG Mobile Agent system Interoperability Facility. I Rothermel og Hohl [RH98], side 50–67.

[MGW97] Dejan S. Milojičić, Shai Guday og Richard Wheeler. Old Wine in New Bottles, Applying OS Process Migration Technology to Mobile Agents. I *3rd. Workshop on Mobile Object Systems, 11th European Conference on Object-Oriented Programming*, juni 1997.

[Moc87] P. Mockapetris. Domain names - concepts and facilities. Rapport, ISI, Network Working Group, november 1987.

[Nee93] Roger M. Needham. Names. I Sape Mullender, redaktør, *Distributed Systems*, kapittel 12, side 315–327. Addison Wesley, 1993.

[Nwa96] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, oktober/november 1996.

[OH98] Robert Orfali og Dan Harkey. *Client/Server Programming with Java and CORBA*. John Wiley and Sons, 2. utgave, januar 1998.

[O'N97] Don O'Neill. Software maintenance and global competitiveness. *Journal of Software Maintenance: Research and Practice*, 9:379–399, 1997.

[PC86] David Lorge Parnas og Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–7, 1986.

[RH98] Kurt Rothermel og Fritz Hohl, redaktører. *Mobile Agents*, bind 1477 av *Lecture Notes in Computer Science*. Springer-Verlag, september 1998.

[RWL95] Trygve Reenskaug, Per Wold og Odd Arild Lehne. *Working With Objects*. Manning Publications Co., 1995.

[Som96] Ian Sommerville. *Software Engineering*. Addison Wesley, 5. utgave, 1996.

[Sta97] William Stallings. *Data and Computer Communications*. Prentice-Hall, 5. utgave, 1997.

[Ste98] W. Richard Stevens. *UNIX network programming: Networking APIs: sockets and XTI*, bind 1. Prentice-Hall, 2. utgave, 1998.

[Tan89] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 2. utgave, 1989.

[TG96] Armstrong A. Takang og Penny A. Grubb. *Software Maintenance*. Thomson Computer Press, 1996.

[TSS+97] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall og G.J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, januar 1997.

[TT97] L.L. Thomsen og B. Thomsen. Mobile agents - the new paradigm in computing. *ICL Systems Journal*, side 14–40, mai 1997.

[Uma97] Amjad Umar. *Object-oriented client/server Internet environments*. Prentice-Hall, 1997.

[Ven97] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, november 1997.

[vHPT97] Arthur van Hoff, Hadi Partovi og Tom Thai. The Open Software Description Format (OSD) . Rapport, The World Wide Web Consortium, 1997. http://www.w3.org/TR/NOTE-OSD.html (27.september 1999).

[Vig98] Giovanni Vigna. *Mobile Code Technologies, Paradigms and Applications*. Doktorgradsoppgave, Politecnico di Milano, februar 1998.

[Voy99] ObjectSpace. *ObjectSpace Voyager ORB 3.0 Developer Guide*, 1999. http://www.objectspace.com/products/documentation/orb.pdf.gz (7. oktober 1999).

[VT97] Jan Vitek og Christian Tschudin, redaktører. *Mobile Object Systems*, bind 1222 av *Lecture Notes in Computer Science*. Springer-Verlag, juli 1997.

[WJ95] M. Wooldridge og N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, oktober 1995.

[WPM99] David Wong, Noemi Paciorek og Dane Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102, mars 1999.

[WWWK97] Jim Waldo, Geoff Wyant, Ann Wollrath og Sam Kendall. A Note on Distributed Computing. I Vitek og Tschudin [VT97], side 49–66.

Tillegg A

Dokumentasjon av MORP

Dette vedlegget inneholder forskjellige informasjon om prototypen MORP som ble presentert i kapittel 4. MORP er programmert i Java versjon 1.2. (også kjent som Java 2). I tillegg til Java standard biblioteker, bruker prototypen også diverse klasser fra Voyager[1] (versjon 3.0).

A.1 Hvordan starte MORP?

MORP består av en tjener og en klient. Disse må startes hver for seg. Tjeneren må startes først. Hvis klassefilene er pakket inn i en `jar`-fil, kan programmet startes på denne måten:

```
java -jar morpserver.jar
```

Klienten kan startes på følgende måte:

```
java -jar morpclient.jar <host>
```

A.2 Hvordan kompilere MORP?

For å gjøre det lettere å kompilere MORP brukes `make`. Makefilen for MORP ser slik ut:

```
1  # Makefile for MORP
2  # Per Thomas Jahr 05.03.99
3  # *****************
4  # Toplevel makefile
5  # *****************
6
7  HOME        = /hom/perja
8  JAVAC       = javac
9  JAVACEA     = $(HOME)tmp/bin/javac-ea
10 JIKES       = $(HOME)/bin/jikes
11 JAR         = jar
12 JAVADOC     = javadoc
```

[1] Voyager er tilgjengelig fra `http://www.objectspace.com` (13. oktober 1999).

```
13 STUBBER     = java-stubber
14 RM          = /local/gnu/bin/rm
15
16 # Lister de forskjellige katalogene hvor koden ligger:
17 SRC         = src
18 TOOLS_SRC   = src/morp/resource/tools
19 REPOSITORY_SRC = src/morp/resource/repository
20 GUI_SRC     = src/morp/gui
21 AGENT_SRC   = src/morp/agent
22
23 # Spesifiserer hvor class filene skal ligge:
24 JAVA_CLASS_DIR = $(HOME)/prog/java/classes
25
26 # Spesifiserer hvor javadoc filene skal ligge:
27 JAVADOC_DIR = $(HOME)/hfag/morp/html
28
29 # Loaksjonen til bibliotker
30 JAVA_LIB    = /local/java/jdk1.2/jre/lib/rt.jar
31 VOYAGER_LIB = $(HOME)/voyager3.0/lib/voyager.jar
32
33 # Klassesti
34 JAVA_CLASSPATH = $(JAVA_LIB):$(VOYAGER_LIB):$(SRC)
35
36 # jar-filer
37 JAR_SERVER_FILE = morpserver.jar
38 JAR_CLIENT_FILE = morpclient.jar
39
40 JAR_SERVER_MANIFEST = MainServerClass
41 JAR_CLIENT_MANIFEST = MainClientClass
42
43 #
44 # rules
45 #
46
47 .PHONY: all noop
48
49 noop:
50       @echo No action taken. Specify target,
51       @echo \'jar\', \'jijar\', \'jejar\', \'javadoc\' or \'clean\'.
52
53 jijar: JAVAC = $(JIKES)
54 jijar: jar
55
56 eajar: JAVAC = $(JAVACEA)
57 eajar: jar
58
59 clean:
60       $(RM) -fr $(JAVA_CLASS_DIR)/*
61       find . -name '*~' -exec del {} \;
62
63 jar:
64       @echo \*\*\*\* Making server jar file
65       $(RM) -fr $(JAVA_CLASS_DIR)/*
66       $(JAVAC) -classpath $(JAVA_CLASSPATH) -d $(JAVA_CLASS_DIR) \
67       $(SRC)/morp/MorpServer.java
68       $(JAR) cfm $(JAR_SERVER_FILE) $(JAR_SERVER_MANIFEST) -C $(JAVA_CLASS_DIR) morp
69       @echo \*\*\*\* Making client jar file
70       $(RM) -fr $(JAVA_CLASS_DIR)/*
71       $(JAVAC) -classpath $(JAVA_CLASSPATH) -d $(JAVA_CLASS_DIR) \
72        $(SRC)/morp/MorpClient.java
73       $(JAR) cfm $(JAR_CLIENT_FILE) $(JAR_CLIENT_MANIFEST) -C $(JAVA_CLASS_DIR) morp
74
```

```
75 javadoc:
76       $(JAVADOC) -d $(JAVADOC_DIR) -private -author -version -sourcepath src/ \
77       morp morp.gui morp.resource.tools morp.resource.repository morp.agent
```

A.3 Manifestfiler

Ved pakking av MORP i `jar`-filer, må `main` metoden samt en gyldig klassesti spesifiseres. Manifestfilene ser slik ut:

Manifest for tjeneren

```
1 Main-Class: morp.MorpServer
2 Class-Path: voyager.jar
```

Manifest for klienten

```
1 Main-Class: morp.MorpClient
2 Class-Path: voyager.jar
```

A.4 Morp.java

```
1  package morp;
2
3  // java imports
4  import java.io.Serializable;
5  import java.util.Calendar;
6  import java.util.Date;
7  import java.util.Hashtable;
8  import java.util.Vector;
9  import java.util.Enumeration;
10
11 // morp imports
12 import morp.resource.tools.ReservationList;
13 import morp.resource.tools.Reservation;
14 import morp.resource.tools.TimeInterval;
15 import morp.resource.tools.TimeIntervalException;
16 import morp.resource.repository.Room;
17 import morp.resource.repository.Person;
18 import morp.Organisation;
19 import morp.gui.MorpFrame;
20
21 /**
22  * Morp class - contains a reference to the reservationlist
23  * This is the server side. Morp will export different interfaces.
24  *
25  * @author  Per Thomas Jahr <perja@ifi.uio.no> 5. 3.1999
26  * @version $Id: Morp.java,v 1.1.1.1.4.5 1999/08/05 19:02:06 perja Exp $
27  */
28 public class Morp implements IMorp, Serializable
29 {
30    // a list of available organisations
31    private static Hashtable myOrganisations = null;
32
33    // a list of clients connected to server
```

```
34    private static Vector myClients = null;
35
36    /**
37     * Constructor
38     */
39    public Morp()
40    {
41       // get an old reservation list ??? read from file?
42       if ( myOrganisations == null )
43       {
44          myOrganisations = new Hashtable();
45       }
46
47       myClients = new Vector();
48
49       // make two organisations
50       Organisation nrOrg = new Organisation( "NR");
51       Organisation sdsOrg = new Organisation( "SDS");
52       Organisation ifiOrg = new Organisation( "IFI");
53
54       addOrganisation( nrOrg );
55       addOrganisation( sdsOrg );
56       addOrganisation( ifiOrg );
57
58       //             roomname org size
59       Room r1 = new Room( "GAMMA", nrOrg, 5 );
60       Room r2 = new Room( "BETA", nrOrg, 7 );
61       Room r3 = new Room( "ALFA", nrOrg, 18 );
62       Room r4 = new Room( "Jotun", sdsOrg, 13 );
63       Room r5 = new Room ( "1305", ifiOrg, 15 );
64
65       Person p1 = new Person( "Thor", nrOrg );
66       Person p2 = new Person( "Arne-Kristian", nrOrg );
67       Person p3 = new Person( "Mr. X", sdsOrg );
68       Person p4 = new Person( "Per Thomas", ifiOrg );
69
70       // add some rooms and persons to the organisations
71       nrOrg.addRoom( r1 );
72       nrOrg.addRoom( r2 );
73       nrOrg.addRoom( r3 );
74       sdsOrg.addRoom( r4 );
75       ifiOrg.addRoom( r5 );
76
77       nrOrg.addPerson( p1 );
78       nrOrg.addPerson( p2 );
79       sdsOrg.addPerson( p3 );
80       ifiOrg.addPerson( p4 );
81    }
82
83    public void addOrganisation( Organisation organisation )
84    {
85       myOrganisations.put( organisation.getName(), organisation );
86    }
87
88    public Organisation[] getOrganisations()
89    {
90       Enumeration orgEnum = myOrganisations.elements();
91       int size = myOrganisations.size();
92       Organisation result[] = new Organisation[size];
93
94       for ( int i = 0; i < size; i++ )
95       {
```

```
96          result[i] = (Organisation)orgEnum.nextElement();
97       }
98
99       return result;
100   }
101
102   public Organisation getOrganisation( String name )
103   {
104      return (Organisation)myOrganisations.get( name );
105   }
106
107   public String[] getOrganisationNames()
108   {
109      Enumeration orgEnum = myOrganisations.keys();
110      int length = myOrganisations.size();
111
112      String name[] = new String[length];
113
114      for ( int i = 0; i < length; i++ )
115      {
116         name[i] = (String)orgEnum.nextElement();
117      }
118
119      return name;
120   }
121
122   public Person findPerson( String personName )
123   {
124      Organisation org[] = getOrganisations();
125      Person personFound = null;
126
127      for ( int i = 0; i < org.length; i++ )
128      {
129         personFound = org[i].findPerson( personName );
130         if ( personFound != null )
131         {
132            // Person is found in one organisation - stop searching
133            break;
134         }
135      }
136
137      return personFound;
138   }
139
140   public Room findRoom( String roomName )
141   {
142      Organisation org[] = getOrganisations();
143      Room roomFound = null;
144
145      for ( int i = 0; i < org.length; i++ )
146      {
147         roomFound = org[i].findRoom( roomName );
148         if ( roomFound != null )
149         {
150            // Room is found in one of the organisations - stop searching
151            break;
152         }
153      }
154
155      return roomFound;
156   }
157
```

```
158    public void registerClient( String name, String port )
159    {
160       myClients.add( "//"+ name + ":"+ port );
161    }
162
163    public boolean signoffClient( String name, String port )
164    {
165       String signoffClient = ( "//"+ name + ":"+ port );
166
167       // remove the object from the vector
168       return myClients.remove( signoffClient );
169    }
170
171    public String[] getConnectedClients()
172    {
173       String resultArray[] = null;
174
175       if ( !myClients.isEmpty() )
176       {
177          resultArray = new String[ myClients.size() ];
178          Enumeration enum = myClients.elements();
179          for ( int i = 0; i < myClients.size(); i++ )
180          {
181             resultArray[i] = (String)enum.nextElement();
182          }
183       }
184       return resultArray;
185    }
186 }
```

A.5 Organisation.java

```
1  package morp;
2
3  // java imports
4  import java.io.Serializable;
5  import java.util.Hashtable;
6  import java.util.Enumeration;
7
8  // morp imports
9  import morp.resource.tools.ReservationList;
10 import morp.resource.repository.Room;
11 import morp.resource.repository.Person;
12
13 /**
14  * Organisation - contains resources (person and room) and resourece
15  * administration tools.
16  *
17  * @author  Per Thomas Jahr <perja@ifi.uio.no> 9. 3.1999
18  * @version $Id: Organisation.java,v 1.1.1.1.4.1 1999/05/05 14:29:50 perja Exp $
19  */
20 public class Organisation implements IOrganisation, Serializable
21 {
22    private String myName = null;
23
24    private Hashtable myRooms = new Hashtable();
25    private Hashtable myPersons = new Hashtable();
26    private Hashtable myResourceTypes = new Hashtable();
27
28    /**
29     * Constructor
```

```
30      */
31     public Organisation( String name )
32     {
33        myName = name;
34     }
35
36     public String getName()
37     {
38        return myName;
39     }
40
41     /**
42      * add a room to this organisation
43      *
44      * @param      room
45      */
46     public void addRoom( Room room )
47     {
48        myRooms.put( room.getName(), room );
49     }
50
51     public void addPerson( Person person )
52     {
53        myPersons.put( person.getName(), person );
54     }
55
56     public Room[] getRooms()
57     {
58        Room room[] = null;
59
60        // Check if hashtable is empty
61        if ( !myRooms.isEmpty() )
62        {
63           Enumeration roomsEnum = myRooms.elements();
64           int myRoomsSize = myRooms.size();
65           room = new Room[ myRoomsSize ];
66
67           for ( int i = 0; i < myRoomsSize ; i++ )
68           {
69              room[i] = (Room)roomsEnum.nextElement();
70           }
71        }
72
73        return room;
74     }
75
76     public String[] getRoomNames()
77     {
78        Enumeration roomEnum = myRooms.keys();
79        int size = myRooms.size();
80        String names[] = new String[size];
81
82        for ( int i = 0; i < size; i++ )
83        {
84           names[i] = (String)roomEnum.nextElement();
85        }
86
87        return names;
88     }
89
90     public Person[] getPersons()
91     {
```

```
92        Person person[] = null;
93
94        // Check if hashtable is empty
95        if ( !myPersons.isEmpty() )
96        {
97           Enumeration personsEnum = myPersons.elements();
98           int myPersonsSize = myPersons.size();
99           person = new Person[ myPersonsSize ];
100
101          for ( int i = 0; i < myPersonsSize; i++ )
102          {
103             person[i] = (Person)personsEnum.nextElement();
104          }
105       }
106       return person;
107    }
108
109    public Person findPerson( String personName )
110    {
111       Person person = (Person)myPersons.get( personName );
112
113       return person;
114    }
115
116    public Room findRoom( String roomName )
117    {
118       Room room = (Room)myRooms.get( roomName );
119
120       return room;
121    }
122
123    public String[] getPersonNames()
124    {
125       Enumeration personEnum = myPersons.keys();
126       int size = myPersons.size();
127       String names[] = new String[size];
128
129       for ( int i = 0; i < size; i++ )
130       {
131          names[i] = (String)personEnum.nextElement();
132       }
133
134       return names;
135    }
136 }
```

A.6 MorpClient.java

```
1  package morp;
2
3  // java imports
4  import java.net.InetAddress;
5  import java.io.File;
6  import java.net.URL;
7
8  // voyager imports
9  import com.objectspace.voyager.Voyager;
10 import com.objectspace.voyager.Namespace;
11 import com.objectspace.lib.util.Console;
12 import com.objectspace.voyager.loader.VoyagerClassLoader;
13 import com.objectspace.voyager.loader.URLResourceLoader;
```

```
14
15 // morp imports
16 import morp.gui.MorpFrame;
17 import morp.resource.tools.IReservationList;
18
19 /**
20  * MorpClient - contact server and start gui
21  *
22  * @author  Per Thomas Jahr <perja@ifi.uio.no> 26. 4.1999
23  * @version $Revision: 1.1.4.13 $
24  */
25 public class MorpClient
26 {
27    // the hostname of this computer
28    private static String myClientName = null;
29
30    // the port number of this client
31    private static String myPort = "10000";
32
33    // the hostname of the MORP server
34    private static String myServerName = null;
35
36    // the file name of the properties file
37    private static String propertiesFileName = "morp_properties";
38
39    // flag for hiding GUI - faster debugging
40    private static boolean showGui = true;
41
42    private static IMorp myIMorp = null;
43    private static IReservationList myReservationList = null;
44
45    public static void main( String argv[] )
46    {
47       // checking arguments
48       if ( argv.length < 1 )
49       {
50          System.out.println( "Error: name of server missing");
51          System.exit(1);
52       }
53
54       // the "-nogui" hides the client window
55       if ( (argv.length == 2) && (argv[0].equals( "-nogui")) )
56       {
57          showGui = false;
58          myServerName = argv[1];
59       }
60       else
61       {
62          myServerName = argv[0];
63       }
64
65       try
66       {
67          // starting a voyager sever on port 10000
68          Voyager.startup( myPort );
69
70          // set log level
71          Console.setLogLevel( Console.VERBOSE );
72
73          // obtain a proxy to the object on port 9000 with symbol "morp"
74          myIMorp = (IMorp) Namespace.lookup( "//"+ myServerName + ":9000/morp");
75
```

```
76          // obtain a proxy to the object on port 9000 with symbol "reservationList"
77          myReservationList = (IReservationList)
78             Namespace.lookup( "//"+ myServerName + ":9000/reservationList");
79
80          // get the local hostname
81          InetAddress localHost = InetAddress.getLocalHost();
82          myClientName = localHost.getHostName();
83
84          // register myself
85          myIMorp.registerClient( myClientName, myPort );
86       }
87       catch( Exception exception )
88       {
89          System.err.println( exception );
90          System.exit(1);
91       }
92
93       // set the resource loader for this client
94       try
95       {
96          URL url = new URL( "http://"
97                        + getServerName()
98                        + ":"
99                        + "9000");
100
101         URLResourceLoader loader = new URLResourceLoader( url );
102         VoyagerClassLoader.addResourceLoader( loader );
103      }
104      catch ( Exception e1 )
105      {
106         e1.printStackTrace();
107      }
108
109      // cheking for properties file
110      File propertiesFile = new File( propertiesFileName );
111
112      if ( !propertiesFile.exists() )
113      {
114         System.err.println( "Error: properties file do not exist");
115         System.exit(1);
116      }
117
118      if ( showGui )
119      {
120         System.out.println( "Starting GUI");
121         MorpFrame morpFrame = new MorpFrame( "MORP");
122      }
123   }
124
125   public static IMorp getMorpInstance()
126   {
127      return myIMorp;
128   }
129
130   public static IReservationList getReservationListInstance()
131   {
132      return myReservationList;
133   }
134
135   public static String getClientName()
136   {
137      return myClientName;
```

```
138    }
139
140    public static String getClientPort()
141    {
142       return myPort;
143    }
144
145    public static String getServerName()
146    {
147       return myServerName;
148    }
149 }
```

A.7 MorpServer.java

```
 1 package morp;
 2
 3 // java imports
 4 import java.net.InetAddress;
 5
 6 // voyager imports
 7 import com.objectspace.voyager.Voyager;
 8 import com.objectspace.voyager.Proxy;
 9 import com.objectspace.voyager.Namespace;
10 import com.objectspace.voyager.ClassManager;
11 import com.objectspace.lib.util.Console;
12
13 // morp imports
14 import morp.gui.MorpConsole;
15 import morp.resource.tools.ReservationList;
16 import morp.resource.tools.IReservationList;
17
18 /**
19  * MorpServer
20  *
21  * @author  Per Thomas Jahr <perja@ifi.uio.no> 26. 4.1999
22  * @version $Revision: 1.1.2.10 $
23  */
24 public class MorpServer
25 {
26    // the servers instance of morp
27    private static Morp myMorp = null;
28
29    // the server name
30    private static String myServerName = null;
31
32    // my port number
33    private static String myPort = "9000";
34
35    public static void main(String[] args)
36    {
37       // starting voyager server and exporting interfaces
38       try
39       {
40          // starting server
41          Voyager.startup( myPort );
42
43          // set log level
44          Console.setLogLevel( Console.VERBOSE );
45
46          // enable HTTP class serving from voyager
```

```
47          ClassManager.enableResourceServer();
48
49          // exporting morp interface
50          myMorp = new Morp();
51          IMorp morp = (IMorp) Proxy.export( myMorp, "9000");
52
53          // bind "MORP-IFI" to this inteface
54          Namespace.bind( "9000/morp", morp );
55
56          // exporting the ReservationList object
57          IReservationList reservationList =
58             (IReservationList)
59             Proxy.export( ReservationList.getInstance(), "9000");
60
61          // bind "MORP-IFI" to this interface
62          Namespace.bind( "9000/reservationList", reservationList );
63
64          // get server name
65          InetAddress localHost = InetAddress.getLocalHost();
66          myServerName = localHost.getHostName();
67       }
68       catch ( Exception ex1 )
69       {
70          System.err.println( ex1 );
71       }
72
73       System.out.println( "Starting MORP console");
74       MorpConsole console = new MorpConsole( "MORP CONSOLE");
75       console.pack();
76       console.show();
77
78       System.out.println( "Server ready!");
79    }
80
81    public static Morp getMorpInstance()
82    {
83       return myMorp;
84    }
85
86    public static String getServerName()
87    {
88       return myServerName;
89    }
90
91    public static String getServerPort()
92    {
93       return myPort;
94    }
95 }
```

A.8 agent/PlaceException.java

```
1 package morp.agent;
2
3 /**
4  * PlaceException - thrown by Upgradeagent if there is no more places
5  * to visit.
6  *
7  * @author  Per Thomas Jahr <perja@ifi.uio.no> 16. 6.1999
8  * @version $Id$
9  **/
```

```
10 class PlaceException extends Exception
11 {
12    public PlaceException()
13    {
14       System.err.println( "PlaceException thrown");
15    }
16 }
```

A.9 agent/UpgradeAgent.java

```
1  package morp.agent;
2
3  // java imports
4  import java.awt.Frame;
5  import java.awt.Label;
6  import java.awt.Button;
7  import java.net.URL;
8  import java.io.Serializable;
9  import java.io.FileInputStream;
10 import java.io.IOException;
11 import java.util.Properties;
12
13 // morp imports
14 import morp.MorpServer;
15 import morp.gui.AddAgendaButton;
16 import morp.gui.MorpFrame;
17 import morp.resource.tools.AgendaList;
18 import morp.resource.tools.IAgendaList;
19
20 // voyager imports
21 import com.objectspace.voyager.Factory;
22 import com.objectspace.voyager.Namespace;
23 import com.objectspace.voyager.Proxy;
24 import com.objectspace.voyager.loader.ArchiveResourceLoader;
25 import com.objectspace.voyager.loader.URLResourceLoader;
26 import com.objectspace.voyager.agent.Agent;
27 import com.objectspace.voyager.mobility.Mobility;
28
29 public class UpgradeAgent implements IUpgradeAgent, Serializable
30 {
31    // an array with the adresses of the places to visit
32    // the adress has following format: //<full machine name>:<port>
33    private String placesToVisit[];
34
35    // keeps track of the agents current location
36    private int currentPlace = 0;
37
38    // number of places to visit
39    private int numberOfPlaces = 0;
40
41    private int itineraryIndex = 0;
42    private int itineraryCount = 0;
43    private String agentResult[];
44    private int resultIndex = 0;
45
46    /**
47     * Constructor
48     */
49    public UpgradeAgent()
50    {
51       System.out.println( "UpgradeAgent created");
```

```
52
53          // register the agenda service
54          try
55          {
56             // exporting the IAgendaList object
57             IAgendaList agendaList =
58                (IAgendaList) Proxy.export( AgendaList.getInstance(), "9000");
59
60             // bind "9000/agendaList" to this interface
61             Namespace.bind( "9000/agendaList", agendaList );
62          }
63          catch ( Exception ex1 )
64          {
65             // FIX ME: Catching all exceptions!
66             System.err.println( ex1 );
67          }
68       }
69
70       public void createTour( String urls[] )
71       {
72          placesToVisit = urls;
73          numberOfPlaces = placesToVisit.length;
74
75          // creating a place to hold results
76          agentResult = new String[ numberOfPlaces ];
77
78          System.out.println( "createTour: places to visit: "+ numberOfPlaces );
79       }
80
81       /**
82        * Returns the next place to visit. Throws exception if there is
83        * no more places to visit.
84        */
85       private String getNextPlace() throws PlaceException
86       {
87          int oldPlace = currentPlace;
88
89          if ( currentPlace + 1 > numberOfPlaces )
90          {
91             // we are already at the last place - throw an exception
92             throw new PlaceException();
93          }
94          else
95          {
96             currentPlace++;
97             System.out.println( "getNextPlace: "+ oldPlace );
98             return placesToVisit[ oldPlace ];
99          }
100      }
101
102      public void go()
103      {
104         /* get next place to visit */
105         String nextPlace = null;
106         try
107         {
108            // set resource loader
109            URL url = new URL( "http://"
110                           + MorpServer.getServerName()
111                           + ":"
112                           + MorpServer.getServerPort() );
113
```

```
114          URLResourceLoader loader = new URLResourceLoader( url );
115          Agent.of( this ).setResourceLoader( loader );
116
117          nextPlace = getNextPlace();
118          // move to client
119          System.out.println( "Move agent to new location: "+ nextPlace );
120
121          // move the agent
122          Agent.of( this ).moveTo( nextPlace, "doWork");
123       }
124       catch ( PlaceException pe )
125       {
126          System.err.println( "Last location reached");
127       }
128       catch( Exception e )
129       {
130          System.err.println( "UpgradeAgent: "+ e );
131          e.printStackTrace();
132       }
133    }
134
135    /**
136     * This method represents the agents work at each client
137     */
138    public void doWork()
139    {
140
141       // looking at preferences
142       System.out.println( "Hi, I'm here");
143       System.out.println( "Analyzing user preferences...");
144
145       Properties properties = new Properties();
146
147       try
148       {
149          FileInputStream inStream = new FileInputStream( "morp_properties");
150          properties.load( inStream );
151       }
152       catch ( IOException ioe )
153       {
154          ioe.printStackTrace();
155       }
156
157       String docking = properties.getProperty( "Allow.agentdocking");
158       agentResult[resultIndex] = docking;
159       resultIndex++;
160
161       if ( docking.equals( "1") )
162       {
163          System.out.println( "Docking is allowed");
164       }
165
166       // adding a new function
167       MorpFrame.getInstance().addButton( new AddAgendaButton() );
168
169       go();
170    }
171
172    /**
173     * This is an example. This method should be more generic.
174     */
175    public void addButton()
```

```
176    {
177       // create a button
178       Button button = new Button( "Status");
179    }
180
181    /**
182     * Returns result from agent tour
183     */
184    public String[] getResults()
185    {
186       return agentResult;
187    }
188 }
```

A.10 gui/AddAgendaButton.java

```
 1 package morp.gui;
 2
 3 // java imports
 4 import java.awt.Button;
 5 import java.awt.event.ActionListener;
 6 import java.awt.event.ActionEvent;
 7
 8 // morp imports
 9 import morp.MorpClient;
10 import morp.resource.tools.IAgendaList;
11 import morp.resource.tools.IReservationList;
12
13 // voyager imports
14 import com.objectspace.voyager.Namespace;
15
16 /**
17  * AddAgendaButton - this class represents a button and it's
18  * actionlistener.
19  *
20  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 8.1999
21  * @version
22  * $Id: AddAgendaButton.java,v 1.1.2.2 1999/08/11 18:03:12 perja Exp $
23  */
24 public class AddAgendaButton extends Button
25 {
26    // a reference to the agendaList
27    private static IAgendaList myAgendaList = null;
28
29    /**
30     * Constructor
31     */
32    public AddAgendaButton()
33    {
34       super( "Add agenda");
35
36       // add a actionlistener for the button
37       addActionListener( new AddAgendaButtonAction() );
38
39       // lookup agendaList
40       try
41       {
42          // obtain a proxy to the object on port 9000 with symbol "agendaList"
43          myAgendaList = (IAgendaList)
44             Namespace.lookup( "//"
45                          + MorpClient.getServerName()
```

```
46                          + ":9000/agendaList");
47          System.out.println( "AddAgendaButton: agendaList lookup OK ("
48                          + MorpClient.getClientName() + ")");
49       }
50       catch ( Exception e1 )
51       {
52          System.err.println( e1 );
53       }
54    }
55
56    /**
57     * A method to get a reference to the agendaList instance
58     */
59    public static IAgendaList getAgendaList()
60    {
61       return myAgendaList;
62    }
63
64    /**
65     * Inner class for "add agenda" button. When the button is pushed,
66     * the following will happen:
67     *  1. get a reference to the selected reservation
68     *  2. show a new dialog/frame for writing a agenda
69     *  3. add this agenda to the agendalist
70     *
71     */
72    class AddAgendaButtonAction implements ActionListener
73    {
74       public void actionPerformed( ActionEvent e )
75       {
76          // start agenda dialog (but not if there is no reservations)
77          IReservationList iresList = MorpClient.getReservationListInstance();
78          int size = iresList.getReservations().length;
79
80          if ( size != 0 )
81          {
82             AgendaDialog agendaDialog = new AgendaDialog( MorpFrame.getInstance() );
83             agendaDialog.show();
84          }
85
86       }
87    }
88 }
```

A.11 gui/AgendaDialog.java

```
1  package morp.gui;
2
3  // java imports
4  import java.awt.BorderLayout;
5  import java.awt.Button;
6  import java.awt.Dialog;
7  import java.awt.Frame;
8  import java.awt.List;
9  import java.awt.TextArea;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;
12 import java.awt.event.ItemListener;
13 import java.awt.event.ItemEvent;
14
15 // morp imports
```

```
16 import morp.MorpClient;
17 import morp.resource.tools.Agenda;
18 import morp.resource.tools.IAgendaList;
19 import morp.resource.tools.IReservationList;
20 import morp.resource.tools.Reservation;
21
22 /**
23  * AgendaDialog - in this dialog the user can write an agenda.
24  *
25  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 8.1999
26  * @version $Id: AgendaDialog.java,v 1.1.2.2 1999/08/11 18:03:12 perja Exp $
27  */
28 public class AgendaDialog extends Dialog
29 {
30    // a local copy of the reservations;
31    private Reservation res[];
32
33    // a list with reservations (GUI)
34    List resList = null;
35
36    // a textarea for the agenda (GUI)
37    TextArea agendaText = null;
38
39    // the index of the previos index
40    int prevIndex = 0;
41
42    /**
43     * Constructor
44     */
45    public AgendaDialog( Frame parent )
46    {
47       // set this dialog modal
48       super( parent, true );
49
50       // get all reservations and show them in a scroll window (list)
51       resList = new List( 5, false );
52       resList.addItemListener( new ResListItemListener() );
53
54       IReservationList iresList = MorpClient.getReservationListInstance();
55       res = iresList.getReservations();
56
57       int size = res.length;
58       System.out.println( "AgendaDialog: size of reservation list: "+ size );
59
60       resList.removeAll();
61
62       for ( int i = 0; i < size; i++ )
63       {
64          // AWT bug - println required ???
65          // System.out.println( "AgendaDialog: i: " + i );
66          resList.add( res[i].getPerson().getName() + " "
67                    + res[i].getRoom().getName() + " "
68                    + res[i].getStartTime() );
69       }
70
71       // select the first reservation in the list
72       resList.select( 0 );
73
74       // agenda textarea
75       agendaText = new TextArea( 10, 15 );
76
77       // set the initial text
```

```
78          showAgenda();
79
80          // quit button
81          Button quitButton = new Button( "Quit");
82          quitButton.addActionListener( new QuitButtonAction() );
83
84          setLayout( new BorderLayout() );
85          add( resList, BorderLayout.NORTH );
86          add( agendaText, BorderLayout.CENTER );
87          add( quitButton, BorderLayout.SOUTH );
88
89          this.pack();
90       }
91
92       /**
93        * A method for finding the index of the selected reservation object
94        *
95        * @return int the index of the reservation object selected in the resList
96        */
97       private int getReservationIndex()
98       {
99          return resList.getSelectedIndex();
100      }
101
102      /**
103       * Save the agenda for a selected index
104       */
105      private void saveAgendaText( int index )
106      {
107         // make an new agenda
108         // get the text from the textArea
109         String text = AgendaDialog.this.agendaText.getText();
110         Agenda agenda = new Agenda( res[ index ], text );
111
112         // add myself to the AgendaList
113         IAgendaList agendaList = AddAgendaButton.getAgendaList();
114         agendaList.addAgenda( res[ index ], agenda );
115
116         System.out.println( "AgendaDialog: Agenda saved");
117      }
118
119      /**
120       * Find and show an agenda
121       */
122      private void showAgenda()
123      {
124         // get the reservation index
125         int index = AgendaDialog.this.getReservationIndex();
126
127         // find the corresponding agenda
128         IAgendaList agendaList = AddAgendaButton.getAgendaList();
129         Agenda agenda = agendaList.getAgenda( AgendaDialog.this.res[ index ] );
130
131         // clear the textarea and show the new agenda
132         if ( agenda != null )
133         {
134            AgendaDialog.this.agendaText.setText( null );
135            AgendaDialog.this.agendaText.setText( agenda.getAgendaString() );
136         }
137         else
138         {
139            AgendaDialog.this.agendaText.setText( "No agenda for this reservation!");
```

```
140         }
141         // update the previous index
142         AgendaDialog.this.prevIndex = index;
143      }
144
145      /**
146       * Inner class - taking care of actions for the quit button
147       */
148      class QuitButtonAction implements ActionListener
149      {
150         public void actionPerformed( ActionEvent e )
151         {
152            // get the reservation index
153            int index = AgendaDialog.this.getReservationIndex();
154
155            // save the last selected index
156            AgendaDialog.this.saveAgendaText( index );
157
158            // close the window
159            AgendaDialog.this.dispose();
160         }
161      }
162
163      /**
164       * Inner class taking care of actions for the list. This action is
165       * activated when a item is clicked. Show a new agenda for this
166       * item.
167       */
168      class ResListItemListener implements ItemListener
169      {
170         public void itemStateChanged( ItemEvent e )
171         {
172            // save the agenda text of the previous reservation
173            saveAgendaText( AgendaDialog.this.prevIndex );
174
175            // show the agenda
176            AgendaDialog.this.showAgenda();
177         }
178      }
179   }
```

A.12 gui/CancelDialog.java

```
1  package morp.gui;
2
3  // java imports
4  import java.awt.Button;
5  import java.awt.Dialog;
6  import java.awt.Frame;
7  import java.awt.GridLayout;
8  import java.awt.Label;
9  import java.awt.List;
10 import java.awt.Panel;
11 import java.awt.TextField;
12 import java.awt.TextArea;
13 import java.awt.event.ActionListener;
14 import java.awt.event.ActionEvent;
15 import java.awt.event.ItemListener;
16 import java.awt.event.ItemEvent;
17
18 // morp imports
```

```
19 import morp.Morp;
20 import morp.IMorp;
21 import morp.MorpClient;
22 import morp.resource.repository.Person;
23 import morp.resource.tools.ReservationList;
24 import morp.resource.tools.IReservationList;
25 import morp.resource.tools.Reservation;
26
27 /**
28  * CancelDialog - search for a reservation and cancel it
29  *
30  * @author  Per Thomas Jahr <perja@ifi.uio.no> 20. 3.1999
31  * @version $Id: CancelDialog.java,v 1.1.1.1.4.4 1999/06/19 15:30:18 perja Exp $
32  */
33 class CancelDialog extends Dialog
34 {
35    private TextField searchField = null;
36    private List resultList = null;
37    private TextArea infoArea = null;
38    private Reservation res[];
39
40    /**
41     * Constructor
42     */
43    public CancelDialog( Frame parent, boolean modal )
44    {
45       // Create a dialog
46       super( parent, modal );
47
48       // Search field for person
49       Label searchLabel = new Label( "Search for person: ");
50       searchField = new TextField( 10 );
51
52       // Result list
53       Label resultLabel = new Label( "Results: ");
54       resultList = new List( 10 );
55       resultList.addItemListener( new ListItemListener() );
56
57       // Buttons
58       Button searchButton = new Button( "Search");
59       Button cancelResButton = new Button( "Cancel reservation");
60       Button quitButton = new Button( "Close");
61
62       searchButton.addActionListener( new SearchButtonAction() );
63       cancelResButton.addActionListener( new CancelResButtonAction() );
64       quitButton.addActionListener( new QuitButtonAction() );
65
66       // Information field (shows info. about selected reservation in list)
67       infoArea = new TextArea( 4, 30 );
68       infoArea.setEditable( false );
69
70       // *** Layout ***
71       setLayout( new GridLayout( 2, 1 ) ); // ( rows, columns )
72
73       Panel leftPanel = new Panel();
74       leftPanel.add( searchLabel );
75       leftPanel.add( searchField );
76       leftPanel.add( searchButton );
77       leftPanel.add( cancelResButton );
78       leftPanel.add( quitButton );
79       leftPanel.add( infoArea );
80
```

```
81        Panel rightPanel = new Panel();
82        rightPanel.add( resultList );
83
84        add( leftPanel );
85        add( rightPanel );
86
87        // setSize( 600, 300 );
88        pack();
89     }
90
91     // Inner class taking care of actions from the list
92     class ListItemListener implements ItemListener
93     {
94        public void itemStateChanged( ItemEvent e )
95        {
96           int index = resultList.getSelectedIndex();
97           String personString = res[index].getPerson().getName();
98           String roomString = res[index].getRoom().getName();
99           String timeString = res[index].getStartTime();
100          String infoString = ( "Person: "+ personString + "\nRoom: "+
101                             roomString + "\nStart time: "+ timeString );
102          infoArea.setText( infoString );
103       }
104    }
105
106    // Inner class taking care of actions for canelreservation button
107    // (cancelResButton)
108    class CancelResButtonAction implements ActionListener
109    {
110       public void actionPerformed( ActionEvent e )
111       {
112          int index = resultList.getSelectedIndex();
113          if ( index == -1 )
114          {
115             infoArea.setText( "Error: No reservation selected.");
116          }
117          else
118          {
119             int number = res[index].getReservationNumber();
120
121             // get interface
122             IReservationList resList = MorpClient.getReservationListInstance();
123
124             // remove reservation
125             resList.remove( number );
126             infoArea.setText( "Reservation removed");
127             resultList.removeAll();
128             CancelDialog.this.dispose();
129          }
130       }
131    }
132
133    // Inner class taking care of actions for search button
134    class SearchButtonAction implements ActionListener
135    {
136       public void actionPerformed( ActionEvent e )
137       {
138          // get interface instance from the MorpClient
139          IReservationList iresList = MorpClient.getReservationListInstance();
140          IMorp morp = MorpClient.getMorpInstance();
141
142          System.out.println( "CancelDialog: got Morp proxy");
```

```
143          // find the person object
144          Person person = morp.findPerson( searchField.getText() );
145          System.out.println( "CancelDialog: got person"+ person.getName() );
146
147          res = iresList.getReservationsForPerson( person );
148          if ( res.length == 0 )
149             System.out.println( "CancelDialog: res has length 0");
150
151          // Remove entries from the list
152          // Do not use the removeAll method if the the list is empty -
153          // because this causes a warning message on UNIX systems
154          // (MOTIF window systems).
155          if ( resultList.getItemCount() != 0 )
156          {
157             resultList.removeAll();
158          }
159
160          // Fill in new entries in the list
161          if ( res.length == 0 )
162          {
163             resultList.add( "No reservations found for "+ searchField.getText() );
164          }
165          else
166          {
167             for ( int i = 0; i < res.length; i++ )
168             {
169                resultList.add( res[i].getStartTime() );
170             }
171          }
172       }
173    }
174
175    // Inner class taking care of actions for quit button
176    class QuitButtonAction implements ActionListener
177    {
178       public void actionPerformed( ActionEvent e )
179       {
180          CancelDialog.this.dispose();
181       }
182    }
183 }
```

A.13 gui/MorpConsole.java

```
1  package morp.gui;
2
3  // java imports
4  import java.awt.Button;
5  import java.awt.Color;
6  import java.awt.Frame;
7  import java.awt.FlowLayout;
8  import java.awt.GridLayout;
9  import java.awt.Label;
10 import java.awt.List;
11 import java.awt.Panel;
12 import java.awt.TextField;
13 import java.awt.event.ActionListener;
14 import java.awt.event.ActionEvent;
15
16 // voyager imports
17 import com.objectspace.voyager.Factory;
```

```
18 import com.objectspace.voyager.Voyager;
19
20 // morp imports
21 import morp.Morp;
22 import morp.MorpServer;
23 import morp.agent.IUpgradeAgent;
24 import morp.agent.UpgradeAgent;
25
26 public class MorpConsole extends Frame
27 {
28    private List placeList = null;
29    private IUpgradeAgent agent = null;
30
31    /**
32     * MorpConsole - constructor for the MORP console
33     */
34    public MorpConsole( String title )
35    {
36       super( title );
37       setLayout( new GridLayout( 2, 1) );
38       setBackground( Color.white );
39
40       // list panel
41       Panel listPanel = new Panel( new GridLayout( 1, 1 ) );
42       placeList = new List( 10 );
43       listPanel.add( placeList );
44
45       // buttons
46       Panel buttonPanel = new Panel ();
47       Button launchButton = new Button( "Launch Agent!");
48       Button showClientsButton = new Button( "Show connected clients");
49       Button resultButton = new Button( "Get results");
50       Button shutdownButton = new Button( "Shutdown");
51
52       buttonPanel.add( launchButton );
53       buttonPanel.add( showClientsButton );
54       buttonPanel.add( resultButton );
55       buttonPanel.add( shutdownButton );
56
57       // add panels
58       this.add( listPanel );
59       this.add( buttonPanel );
60
61       // bind actions to buttons
62       launchButton.addActionListener( new LaunchButtonAction() );
63       showClientsButton.addActionListener( new ShowClientsButtonAction() );
64       resultButton.addActionListener( new ResultButtonAction() );
65       shutdownButton.addActionListener( new ShutdownButtonAction() );
66    }
67
68    /**
69     * LaunchButtonAction - takes care of launch button actions
70     */
71    class LaunchButtonAction implements ActionListener
72    {
73       public void actionPerformed( ActionEvent e )
74       {
75          // create the agent
76          try
77          {
78             agent = (IUpgradeAgent)
79                Factory.create( UpgradeAgent.class.getName() );
```

```
80             agent.createTour( placeList.getItems() );
81             agent.go();
82          }
83          catch( Exception exception )
84          {
85             System.err.println( exception );
86          }
87       }
88    }
89
90    class ShutdownButtonAction implements ActionListener
91    {
92       public void actionPerformed( ActionEvent e )
93       {
94          MorpConsole.this.dispose();
95          Voyager.shutdown();
96          System.exit( 0 );
97       }
98    }
99
100   class ResultButtonAction implements ActionListener
101   {
102      public void actionPerformed( ActionEvent e )
103      {
104         String results[] = agent.getResults();
105         for ( int i = 0; i < results.length; i++ )
106         {
107            System.out.println( "Results: "+ results[i] );
108         }
109      }
110   }
111
112   class ShowClientsButtonAction implements ActionListener
113   {
114      public void actionPerformed( ActionEvent e )
115      {
116         // fill list with places
117         Morp morp = MorpServer.getMorpInstance();
118         String places[] = morp.getConnectedClients();
119
120         if ( places == null )
121         {
122            MorpConsole.this.placeList.add( "No clients connected");
123         }
124         else
125         {
126            MorpConsole.this.placeList.removeAll();
127            for ( int i = 0; i < places.length; i++ )
128            {
129               MorpConsole.this.placeList.add( places[i] );
130            }
131         }
132      }
133   }
134 }
```

A.14 gui/MorpFrame.java

```
1 package morp.gui;
2
3 // java imports
```

```
4  import java.awt.Button;
5  import java.awt.Font;
6  import java.awt.Frame;
7  import java.awt.Panel;
8  import java.awt.Choice;
9  import java.awt.Label;
10 import java.awt.List;
11 import java.awt.GridLayout;
12 import java.awt.event.ActionListener;
13 import java.awt.event.ActionEvent;
14 import java.awt.event.ItemListener;
15 import java.awt.event.ItemEvent;
16
17 // morp imports
18 import morp.IMorp;
19 import morp.MorpClient;
20 import morp.Organisation;
21 import morp.resource.repository.Person;
22 import morp.resource.repository.Room;
23 import morp.resource.tools.ReservationList;
24 import morp.resource.tools.IReservationList;
25 import morp.resource.tools.Reservation;
26
27 // voyager imports
28 import com.objectspace.voyager.Voyager;
29
30 /**
31  * MorpFrame - this is the first window the user will see. Contains
32  * buttons for making and displaying resources.
33  *
34  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 3.1999
35  * @version
36  * $Id: MorpFrame.java,v 1.1.1.1.4.10 1999/08/11 18:03:12 perja Exp $
37  */
38 public class MorpFrame extends Frame
39 {
40    // myInstance (singleton pattern)
41    private static MorpFrame myInstance = null;
42
43    // List
44    private List resList = new List( 5 );
45
46    // Button panel
47    private Panel buttonPanel = new Panel();
48
49    // choices
50    private ResourceChoice orgChoice = null;
51    private ResourceChoice resChoice = null;
52
53    // Constructor
54    public MorpFrame( String title )
55    {
56       // Call superclass with windowtitle as an argument
57       super( title );
58
59       // set my instance
60       myInstance = this;
61
62       // set the layoutmanager ( height, width )
63       setLayout( new GridLayout( 4, 1 ) );
64
65       // Choice panel (north-west)
```

```
66        Panel choicePanel = new Panel( new GridLayout( 2, 2 ) );
67
68        Label orgLabel = new Label( "Show organisation: ");
69
70        // get the names out of the organisation objects
71        IMorp morp = MorpClient.getMorpInstance();
72        String orgNames[] = morp.getOrganisationNames();
73
74        // choices in drop-down list
75        orgChoice = new ResourceChoice( this, orgNames );
76        orgChoice.addItemListener( new ChoiceListener() );
77
78        String resItems[] = { "Rooms", "Persons"};
79
80        resChoice = new ResourceChoice( this, resItems );
81        resChoice.addItemListener( new ChoiceListener() );
82
83        Label resLabel = new Label( "Show resource: ");
84
85        choicePanel.add( orgLabel );
86        choicePanel.add( orgChoice );
87        choicePanel.add( resLabel );
88        choicePanel.add( resChoice );
89
90        // Buttons
91        Button resButton = new Button( "Reservation...");
92        Button cancelResButton = new Button( "Cancel reservation...");
93        Button viewResButton = new Button( "Show all reservations");
94        Button prefsButton = new Button( "Customize MORP...");
95        Button quitButton = new Button( "Quit!");
96
97        resButton.addActionListener( new ResButtonAction() );
98        cancelResButton.addActionListener( new CancelResButtonAction() );
99        viewResButton.addActionListener( new ViewResButtonAction() );
100       prefsButton.addActionListener( new PrefsButtonAction() );
101       quitButton.addActionListener( new QuitButtonAction() );
102
103       buttonPanel.add( resButton );
104       buttonPanel.add( cancelResButton );
105       buttonPanel.add( viewResButton );
106       buttonPanel.add( prefsButton );
107       buttonPanel.add( quitButton );
108
109       // update the list
110       updateList( orgChoice.getSelectedItem(), resChoice.getSelectedItem() );
111
112       // title of this window
113       Font font = new Font( "SansSerif", Font.BOLD, 36 );
114       Label morpLabel = new Label( "Room reservation", Label.CENTER );
115       morpLabel.setFont( font );
116
117       // add the panels
118       add( morpLabel );
119       add( resList );
120       add( choicePanel );
121       add( buttonPanel );
122       pack();
123       show();
124    }
125
126    /**
127     * Gets the only instance of the MorpFrame class
```

```
128      *
129      * @return MorpFrame this instance of the MorpFrame
130      */
131     public static MorpFrame getInstance()
132     {
133        return myInstance;
134     }
135
136     /**
137      * A method for adding buttons to this console.
138      *
139      * @param newButton the new button to add
140      */
141     public void addButton( Button newButton )
142     {
143        // adding the button
144        buttonPanel.add( newButton );
145
146        // the gui should be updated automatically
147     }
148
149     private void updateList( String orgName, String resName )
150     {
151        IMorp morp = MorpClient.getMorpInstance();
152        Organisation org = morp.getOrganisation( orgName );
153        String names[] = null;
154
155        // remove old items
156        resList.removeAll();
157
158        if ( resName.equals( "Persons") )
159        {
160           names = org.getPersonNames();
161           for ( int i = 0; i < names.length; i++ )
162           {
163              resList.add( names[i] );
164           }
165        }
166        else
167        {
168           names = org.getRoomNames();
169           for ( int i = 0; i < names.length; i++ )
170           {
171              resList.add( names[i] );
172           }
173        }
174
175     }
176
177     // Inner class taking care of actions for the drop-down list
178     class ChoiceListener implements ItemListener
179     {
180        public void itemStateChanged( ItemEvent e )
181        {
182           updateList( orgChoice.getSelectedItem(),
183                    resChoice.getSelectedItem() );
184        }
185     }
186
187     // Inner class taking care of actions for reservation button (resButton)
188     class ResButtonAction implements ActionListener
189     {
```

```
190         public void actionPerformed( ActionEvent e )
191         {
192            // Show the reservation window
193            ReservationFrame resFrame = ReservationFrame.getInstance();
194            resFrame.show();
195         }
196      }
197
198      // Inner class taking care of actions for canelreservation button
199      // (cancelResButton)
200      class CancelResButtonAction implements ActionListener
201      {
202         public void actionPerformed( ActionEvent e )
203         {
204            // Show the cancel dialog
205            CancelDialog cancelDialog = new CancelDialog( MorpFrame.this, true );
206            cancelDialog.show();
207         }
208      }
209
210      // Inner class taking care of actions for the reservation view button
211      class ViewResButtonAction implements ActionListener
212      {
213         public void actionPerformed( ActionEvent e )
214         {
215            IReservationList iresList = MorpClient.getReservationListInstance();
216            Reservation res[] = iresList.getReservations();
217            int size = res.length;
218            System.out.println( "Length of reservation list: "+ size );
219
220            // clear list
221            resList.removeAll();
222
223            // fill in information about reservations in list
224            for ( int i = 0; i < size; i++ )
225            {
226               // should use toString ???
227               resList.add( res[i].getPerson().getName() + " "
228                        + res[i].getRoom().getName() + " "
229                        + res[i].getStartTime() );
230            }
231         }
232      }
233
234      // Inner class taking care of actions for preferences button
235      class PrefsButtonAction implements ActionListener
236      {
237         public void actionPerformed( ActionEvent e )
238         {
239            PreferencesDialog prefsDialog = new PreferencesDialog( MorpFrame.this );
240            prefsDialog.show();
241         }
242      }
243
244      // Inner class taking care of actions for quit button
245      class QuitButtonAction implements ActionListener
246      {
247         public void actionPerformed( ActionEvent e )
248         {
249            // tell the server that I'm signing off
250            IMorp iMorp = MorpClient.getMorpInstance();
251
```

```
252          boolean isSignedOff =
253             iMorp.signoffClient( MorpClient.getClientName(),
254                                  MorpClient.getClientPort() );
255
256          if ( isSignedOff )
257          {
258             System.out.println( "Client ("
259                                 + MorpClient.getClientName()
260                                 + ") signed off from server ("
261                                 + MorpClient.getServerName() +")" );
262          }
263          else
264          {
265             System.out.println( "Error: could not sign off from server");
266          }
267
268          // shutdown voyager server
269          Voyager.shutdown();
270
271          // dispose window
272          MorpFrame.this.dispose();
273          System.exit( 0 );
274       }
275    }
276 }
```

A.15 gui/PreferencesDialog.java

```
1  package morp.gui;
2
3  // java imports
4  import java.awt.Button;
5  import java.awt.Checkbox;
6  import java.awt.Dialog;
7  import java.awt.Frame;
8  import java.awt.GridLayout;
9  import java.awt.Label;
10 import java.awt.Panel;
11 import java.awt.event.ActionEvent;
12 import java.awt.event.ActionListener;
13 import java.awt.event.ItemEvent;
14 import java.awt.event.ItemListener;
15 import java.io.FileInputStream;
16 import java.io.FileOutputStream;
17 import java.io.IOException;
18 import java.util.Enumeration;
19 import java.util.Properties;
20
21 /**
22  * PreferencesDialog - the purpose of this dialog is to customize
23  * different settings in MORP. The user can specifiy the upgrade
24  * degree.
25  *
26  * @author  Per Thomas Jahr <perja@ifi.uio.no> 8. 6.1999
27  * @version $Id: PreferencesDialog.java,v 1.1.2.3 1999/06/24 12:22:31 perja Exp $
28  **/
29 class PreferencesDialog extends Dialog
30 {
31    // the properties/preferences
32    private Properties myProperties = null;
33
```

```
34    // original properties
35    private Properties myOrgProperties = null;
36
37    // my checkboxes
38    private Checkbox myCheckboxes[] = null;
39
40    public PreferencesDialog( Frame parent )
41    {
42       // set this dialog modal
43       super( parent, true );
44
45       setLayout( new GridLayout( 3, 1 ) ); // ( rows, columns )
46
47       // loadpreferences
48       myProperties = loadPreferences();
49       myOrgProperties = (Properties)myProperties.clone();
50
51       // make checkboxes
52       Panel checkboxPanel = makeCheckboxPanel( myProperties );
53
54       // make buttons
55       Button resetButton = new Button( "Reset");
56       Button saveButton = new Button( "Save");
57       Button cancelButton = new Button( "Cancel");
58
59       Panel buttonPanel = new Panel();
60
61       buttonPanel.add( resetButton );
62       buttonPanel.add( saveButton );
63       buttonPanel.add( cancelButton );
64
65       resetButton.addActionListener( new ResetButtonAction() );
66       saveButton.addActionListener( new SaveButtonAction() );
67       cancelButton.addActionListener( new CancelButtonAction() );
68
69       // add panels
70       add( new Label( "Preferences: ") );
71       add( checkboxPanel );
72       add( buttonPanel );
73
74       this.pack();
75    }
76
77    /**
78     * Loading preferences from file
79     */
80    Properties loadPreferences()
81    {
82       Properties result = new Properties();
83
84       try
85       {
86          FileInputStream inStream =
87             new FileInputStream( "morp_properties");
88          result.load( inStream );
89       }
90       catch ( IOException ioe )
91       {
92          ioe.printStackTrace();
93       }
94       return result;
95    }
```

```
96
97    /**
98     * Save properties to a file.
99     */
100   void savePreferences( Properties properties )
101   {
102      try
103      {
104         FileOutputStream outStream =
105            new FileOutputStream( "morp_properties");
106         properties.store( outStream, "MORP properties");
107      }
108      catch ( IOException ioe )
109      {
110         ioe.printStackTrace();
111      }
112   }
113
114   /**
115    * Make checkboxes with the specified state
116    */
117   Panel makeCheckboxPanel( Properties properties )
118   {
119      // the checkbox panel
120      Panel checkboxPanel =
121         new Panel( new GridLayout( properties.size(), 1 ) );
122
123      // the state/value of the property
124      boolean state = false;
125
126      // an enumeration of all names in the property list
127      Enumeration enum = properties.propertyNames();
128
129      // create checkboxes
130      Checkbox checkboxes[] = new Checkbox[ properties.size() ];
131
132      // the checkbox itemlistener - adding one object of the class
133      // CheckBoxListener to all checkboxes (inside for-loop)
134      CheckBoxListener listener = new CheckBoxListener();
135
136      for ( int i = 0; i < properties.size(); i++ )
137      {
138         // find the name for this property
139         String name = (String)enum.nextElement();
140
141         // find the state for this property
142         if ( (properties.getProperty( name ).equals( "1")) )
143         {
144            state = true;
145         }
146         else
147         {
148            state = false;
149         }
150
151         // creating a new checkbox
152         checkboxes[i] = new Checkbox( name, state );
153
154         // adding a itemlistener to all checkboxes
155         checkboxes[i].addItemListener( listener );
156
157         // adding the checkbox to the panel
```

```
158            checkboxPanel.add( checkboxes[i] );
159         }
160         myCheckboxes = checkboxes;
161         return checkboxPanel;
162      }
163
164      /**
165       * Inner class - takes care of actions for reset button. Presseing
166       * the reset button will reset all values to default.
167       */
168      class ResetButtonAction implements ActionListener
169      {
170         public void actionPerformed( ActionEvent e )
171         {
172
173         }
174      }
175
176      /**
177       * Inner class - takes care of actions for save button. Pressing
178       * this button will save all settings in this window.
179       */
180      class SaveButtonAction implements ActionListener
181      {
182         public void actionPerformed( ActionEvent e )
183         {
184            savePreferences( PreferencesDialog.this.myProperties );
185            PreferencesDialog.this.dispose();
186         }
187      }
188
189      /**
190       * Inner class - takes care of actions for cancel button. Pressing
191       * this button will restore previous settings and close dialog.
192       */
193      class CancelButtonAction implements ActionListener
194      {
195         public void actionPerformed( ActionEvent e )
196         {
197            PreferencesDialog.this.dispose();
198         }
199      }
200
201      /**
202       * Inner class for checkbox actions
203       */
204      class CheckBoxListener implements ItemListener
205      {
206         public void itemStateChanged( ItemEvent e )
207         {
208            int length = PreferencesDialog.this.myCheckboxes.length;
209
210            for ( int i = 0; i < length; i++ )
211            {
212               Checkbox tmpCheckBox = PreferencesDialog.this.myCheckboxes[i];
213               String state = null;
214
215               if ( e.getSource().equals( tmpCheckBox) )
216               {
217                  if ( tmpCheckBox.getState() )
218                     state = "1";
219                  else
```

```
220                     state = "0";
221
222                 Properties myProperties = PreferencesDialog.this.myProperties;
223                 myProperties.setProperty( tmpCheckBox.getLabel(), state );
224             }
225         }
226       }
227    }
228 }
```

A.16 gui/ReservationFrame.java

```
 1 package morp.gui;
 2
 3 // java imports
 4 import java.awt.Frame;
 5 import java.awt.Button;
 6 import java.awt.Panel;
 7 import java.awt.Choice;
 8 import java.awt.TextField;
 9 import java.awt.Label;
10 import java.awt.GridLayout;
11 import java.awt.BorderLayout;
12 import java.awt.List;
13 import java.awt.event.ActionListener;
14 import java.awt.event.ActionEvent;
15 import java.util.Enumeration;
16 import java.util.Vector;
17 import java.util.Calendar;
18 import java.util.Date;
19 import java.util.TimeZone;
20
21 // morp.gui imports
22 import morp.Morp;
23 import morp.MorpClient;
24 import morp.Organisation;
25 import morp.IMorp;
26 import morp.resource.tools.Reservation;
27 import morp.resource.tools.ReservationList;
28 import morp.resource.tools.IReservationList;
29 import morp.resource.tools.TimeInterval;
30 import morp.resource.tools.TimeIntervalException;
31 import morp.resource.repository.Person;
32 import morp.resource.repository.Room;
33
34 /**
35  * ReservationFrame - this window shold assist the user in setting
36  * up an reservation
37  *
38  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 3.1999
39  * @version $Id: ReservationFrame.java,v 1.1.1.1.4.3 1999/05/06 13:19:07 perja Exp $
40  */
41 public class ReservationFrame extends Frame
42 {
43    private static ReservationFrame myInstance = null;
44
45    private TextField yearField = null;
46    private TextField dateField = null;
47    private TextField hourField = null;
48    private TextField minuteField = null;
49    private TextField durationField = null;
```

```
50
51    private Choice monthChoice = null;
52
53    private List roomList = null;
54    private List personList = null;
55
56    // Constructor
57    private ReservationFrame()
58    {
59       // Set the title
60       super( "Reservation ");
61
62       // set the layout
63       setLayout( new GridLayout( 2, 1 ) );
64
65       Panel datePanel = new Panel();
66       Panel listPanel = new Panel();
67       Panel buttonPanel = new Panel( new GridLayout( 1, 2 ) );
68
69       // get the timezone for this location
70       TimeZone timeZone = TimeZone.getDefault();
71
72       // Get current time and date. Fill them in as default values.
73       Calendar cal = Calendar.getInstance();
74       cal.setTimeZone( timeZone );
75
76       int year = cal.get( Calendar.YEAR );
77       int month = cal.get( Calendar.MONTH );
78       int date = cal.get( Calendar.DATE );
79       int hour = cal.get( Calendar.HOUR_OF_DAY );
80
81       // Fill the date panel
82       Label yearLabel = new Label( "Year: ");
83       yearField = new TextField( Integer.toString( year ), 4 );
84
85       Label monthLabel = new Label( "Month: ");
86       monthChoice = new Choice();
87
88       String months[] = { "January", "February", "March",
89                           "April", "May", "June", "July",
90                           "August", "September", "October",
91                           "November", "December"};
92
93       for ( int i = 0; i < 12; i++ )
94       {
95          monthChoice.add( months[i] );
96       }
97
98       // Set the default for the months
99       monthChoice.select( month );
100
101      Label dateLabel = new Label( "Date: ");
102      dateField = new TextField( Integer.toString( date ), 2 );
103
104      Label hourLabel = new Label( "Hour: ");
105      hourField = new TextField( Integer.toString( hour ), 2 );
106
107      Label minuteLabel = new Label( "Minute: ");
108      minuteField = new TextField( "00", 2 );
109
110      Label durationLabel = new Label( "Duration (hours): ");
111      durationField = new TextField( "1", 1 );
```

```
112
113         datePanel.add( dateLabel );
114         datePanel.add( dateField );
115         datePanel.add( monthLabel );
116         datePanel.add( monthChoice );
117         datePanel.add( yearLabel );
118         datePanel.add( yearField );
119         datePanel.add( hourLabel );
120         datePanel.add( hourField );
121         datePanel.add( minuteLabel );
122         datePanel.add( minuteField );
123         datePanel.add( durationLabel );
124         datePanel.add( durationField );
125
126         // Fill the list panel
127
128         // get the organisations
129         IMorp morp = MorpClient.getMorpInstance();
130         Organisation org[] = morp.getOrganisations();
131         int length = org.length;
132
133         // get the persons and rooms in the organisations
134         Vector personVector = new Vector();
135         Vector roomVector = new Vector();
136
137         for ( int i = 0; i < length; i++ )
138         {
139            String persons[] = null;
140            persons = org[i].getPersonNames();
141
142            for ( int j = 0; j < persons.length; j++ )
143            {
144               personVector.addElement( persons[j] );
145            }
146
147            String rooms[] = null;
148            rooms = org[i].getRoomNames();
149
150            for ( int j = 0; j < rooms.length; j++ )
151            {
152               roomVector.addElement( rooms[j] );
153            }
154         }
155
156         // Convert the two vectors to arrays
157         Enumeration personEnum = personVector.elements();
158         Enumeration roomEnum = roomVector.elements();
159
160         int personLength = personVector.size();
161         int roomLength = roomVector.size();
162
163         String persons[] = new String[personLength];
164         String rooms[] = new String[roomLength];
165
166         for ( int i = 0; i < personLength; i++ )
167         {
168            persons[i] = (String)personEnum.nextElement();
169         }
170
171         for ( int i = 0; i < roomLength; i++ )
172         {
173            rooms[i] = (String)roomEnum.nextElement();
```

```
174       }
175
176       // create lists
177       roomList = new List( 5 );
178       personList = new List( 5 );
179
180       // fill the room list
181       for ( int i = 0; i < roomLength; i++ )
182       {
183
184          roomList.add( rooms[i] );
185       }
186
187       // fill the person list
188       for ( int i = 0; i < personLength; i++ )
189       {
190
191          personList.add( persons[i] );
192       }
193
194       listPanel.add( roomList );
195       listPanel.add( personList );
196
197       // fill the button panel
198
199       Button submitButton = new Button( "Submit");
200       Button quitButton = new Button( "Cancel");
201
202       submitButton.addActionListener( new SubmitButtonAction() );
203       quitButton.addActionListener( new QuitButtonAction() );
204
205       buttonPanel.add( submitButton );
206       buttonPanel.add( quitButton );
207
208       // add panels
209       Panel lowerPanel = new Panel( new BorderLayout() );
210       lowerPanel.add( listPanel, BorderLayout.NORTH );
211       lowerPanel.add( buttonPanel, BorderLayout.SOUTH );
212
213       add( datePanel );
214       add( lowerPanel );
215
216       setSize( 400, 400 );
217    }
218
219    public static ReservationFrame getInstance()
220    {
221       if ( myInstance == null )
222       {
223          // Make a new reservation
224          myInstance = new ReservationFrame();
225       }
226       return myInstance;
227    }
228
229    // Inner class taking care of actions for submit button
230    class SubmitButtonAction implements ActionListener
231    {
232       public void actionPerformed( ActionEvent e )
233       {
234          int year  = 0;
235          int date  = 0;
```

```
236          int hour  = 0;
237          int minute = 0;
238          int duration = 0;
239
240          // create a new timeinterval - no checking
241          Calendar cal = Calendar.getInstance();
242
243          year   = Integer.parseInt( yearField.getText() );
244          date   = Integer.parseInt( dateField.getText() );
245          hour   = Integer.parseInt( hourField.getText() );
246          minute = Integer.parseInt( minuteField.getText() );
247          duration = Integer.parseInt( durationField.getText() );
248
249          int month = monthChoice.getSelectedIndex();
250
251          // find the personobject
252          String personName = personList.getSelectedItem();
253          System.out.println( "Name from list: "+ personName );
254          IMorp morp = MorpClient.getMorpInstance();
255          Person person = morp.findPerson( personName );
256          System.out.println( "Name from name lookup on person "
257                        + personName
258                        + ": "
259                        + person.getName() );
260
261          String roomName = roomList.getSelectedItem();
262          System.out.println( "Room from list: "+ roomName );
263          Room room = morp.findRoom( roomName );
264          System.out.println( "Name from name lookup on room "
265                        + roomName
266                        + ": "
267                        + room.getName() );
268
269          if ( ( person != null ) && ( room != null ) )
270          {
271             // add reservation to reservationList
272             IReservationList resList = MorpClient.getReservationListInstance();
273             System.out.println( "ReservationFrame: got reslist");
274             resList.add( person, room, year, month, date, hour, minute, duration );
275
276             System.out.println( "Reservation added");
277          }
278       }
279    }
280
281    // Inner class taking care of actions for quit button
282    class QuitButtonAction implements ActionListener
283    {
284       public void actionPerformed( ActionEvent e )
285       {
286          // exit
287          myInstance.dispose();
288       }
289    }
290 }
```

A.17 gui/ResourceChoice.java

```
1 package morp.gui;
2
3 // java imports
```

```
4  import java.awt.Choice;
5  import java.awt.Frame;
6  import java.awt.event.ItemListener;
7  import java.awt.event.ItemEvent;
8
9  /**
10  * ResourceChoice - a pulldown list for resources
11  *
12  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 3.1999
13  * @version $Id: ResourceChoice.java,v 1.1.1.1 1999/04/23 07:37:47 perja Exp $
14  */
15 public class ResourceChoice extends Choice
16 {
17    private Frame myParentFrame = null;
18
19    public ResourceChoice( Frame parentFrame, String items[] )
20    {
21       myParentFrame = parentFrame;
22       for ( int counter = 0; counter < items.length; counter++ )
23       {
24          addItem( items[ counter ] );
25       }
26       addItemListener( new ChoiceListener() );
27    }
28
29    // inner class
30    class ChoiceListener implements ItemListener
31    {
32       public void itemStateChanged( ItemEvent e )
33       {
34          // show selected list
35       }
36    }
37 }
```

A.18 resource/repository/Person.java

```
1  package morp.resource.repository;
2
3  // java imports
4  import java.io.Serializable;
5
6  // morp imports
7  import morp.Organisation;
8  import morp.resource.tools.Reservation;
9  import morp.resource.tools.ReservationList;
10
11
12 /**
13  * Person - holds information about each person
14  *
15  * @author  Per Thomas Jahr <perja@ifi.uio.no> 9. 3.1999
16  * @version $Id: Person.java,v 1.1.1.1.4.1 1999/05/05 14:29:51 perja Exp $
17  */
18 public class Person extends Resource implements Serializable
19 {
20    private String myName = null;
21
22    public Person( String name, Organisation org )
23    {
24       super( name, org );
```

```
25    }
26
27    public Reservation[] listReservations()
28    {
29       ReservationList rlist = ReservationList.getInstance();
30       return rlist.getReservationsForPerson( this );
31    }
32 }
```

A.19 resource/repository/Resource.java

```
1  package morp.resource.repository;
2
3  // java imports
4  import java.io.Serializable;
5
6  // morp imports
7  import morp.Organisation;
8  import morp.resource.tools.Reservation;
9
10 /**
11  * Resource class
12  *
13  * @author  Per Thomas Jahr <perja@ifi.uio.no> 5. 3.1999
14  * @version $Id: Resource.java,v 1.1.1.1.4.2 1999/05/30 11:20:11 perja Exp $
15  */
16 public abstract class Resource implements Serializable
17 {
18    private String  myName = null;
19    private Organisation myOrg = null;
20
21    public Resource( String name, Organisation org )
22    {
23       myName = name;
24       myOrg = org;
25    }
26
27    public String getName()
28    {
29       return myName;
30    }
31
32    public Organisation getOrganisation()
33    {
34       return myOrg;
35    }
36
37    public abstract Reservation[] listReservations();
38 }
```

A.20 resource/repository/Room.java

```
1  package morp.resource.repository;
2
3  // java imports
4  import java.io.Serializable;
5
6  // morp imports
7  import morp.Organisation;
8  import morp.resource.tools.Reservation;
```

```
 9
10 /**
11  * Room class
12  *
13  * @author  Per Thomas Jahr <perja@ifi.uio.no> 5. 3.1999
14  * @version $Id: Room.java,v 1.1.1.1.4.1 1999/05/05 14:29:51 perja Exp $
15  */
16 public class Room extends Resource implements Serializable
17 {
18    private int mySize = 0; // size of rooms (eg. chairs)
19
20    public Room( String name, Organisation org, int size )
21    {
22       super( name, org );
23       mySize = size;
24    }
25
26    public int getSize()
27    {
28       return mySize;
29    }
30
31    public Reservation[] listReservations()
32    {
33       // return Morp.getReservationEntrys( super.myName );
34       return null;
35    }
36 }
```

A.21 resource/tools/Agenda.java

```
 1 package morp.resource.tools;
 2
 3 // java imports
 4 import java.io.Serializable;
 5
 6 /**
 7  * Agenda - this class represents a agenda for a meeting.
 8  *
 9  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 8.1999
10  * @version $Id: Agenda.java,v 1.1.2.2 1999/08/11 18:03:13 perja Exp $
11  */
12 public class Agenda implements Serializable
13 {
14    // the reservation associated with this agenda
15    private Reservation myReservation = null;
16
17    // the agenda item
18    private String myAgendaString = null;
19
20    /**
21     * Constructor
22     */
23    public Agenda( Reservation reservation, String agendaString )
24    {
25       myReservation = reservation;
26       myAgendaString = agendaString;
27    }
28
29    /**
30     * Get the reservation associated with this agenda
```

```
31      *
32      * @return Reservation the associated reservation
33      */
34     public Reservation getReservation()
35     {
36        return myReservation;
37     }
38
39     /**
40      * Get the agenda string - the agenda
41      */
42     public String getAgendaString()
43     {
44        return myAgendaString;
45     }
46 }
```

A.22 resource/tools/AgendaList.java

```
1  package morp.resource.tools;
2
3  // java imports
4  import java.io.Serializable;
5  import java.util.Hashtable;
6
7  // morp imports
8  import morp.resource.tools.Reservation;
9
10 /**
11  * AgendaList - this is a holder for all the agendas.
12  *
13  * @author  Per Thomas Jahr <perja@ifi.uio.no> 10. 8.1999
14  * @version $Id: AgendaList.java,v 1.1.2.2 1999/08/11 18:03:13 perja Exp $
15  */
16 public class AgendaList implements IAgendaList, Serializable
17 {
18    // singleton - myInstance
19    private static AgendaList myInstance = null;
20
21    // my agendas
22    private static Hashtable myAgendas = null;
23
24    /**
25     * Constructor - this is private due to singleton
26     */
27    private AgendaList()
28    {
29       myAgendas = new Hashtable();
30    }
31
32    /**
33     * Get instance
34     *
35     * @return AgendaList the only instance of this class
36     */
37    public static AgendaList getInstance()
38    {
39       if ( myInstance == null )
40       {
41          myInstance = new AgendaList();
42       }
```

```
43          return myInstance;
44      }
45
46      /**
47       * Add a agenda to the list
48       *
49       * @param reservation the reservation (index used as key)
50       * @param agenda  the agenda for this reservation
51       */
52      public void addAgenda( Reservation reservation, Agenda agenda )
53      {
54          myAgendas.put( new Integer( reservation.getReservationNumber() ), agenda );
55          System.out.println( "AgendaList: Agenda added: size"+ myAgendas.size() );
56      }
57
58      /**
59       * Find a agenda for a reservation
60       *
61       * @param reservation the reservation to find a agenda for
62       * @return agenda  the agenda corresponding to the reservation or null if no
63       *                 agenda is found.
64       */
65      public Agenda getAgenda( Reservation reservation )
66      {
67          Integer newInteger = new Integer( reservation.getReservationNumber() );
68          return (Agenda)myAgendas.get( newInteger );
69      }
70  }
```

A.23 resource/tools/Reservation.java

```
1   package morp.resource.tools;
2
3   // java imports
4   import java.io.Serializable;
5
6   // morp imports
7   import morp.resource.repository.Room;
8   import morp.resource.repository.Person;
9   import morp.resource.tools.ReservationList;
10  import morp.resource.tools.TimeInterval;
11
12  /**
13   * Reservation
14   *
15   * @author  Per Thomas Jahr <perja@ifi.uio.no> 5. 3.1999
16   * @version $Id: Reservation.java,v 1.1.1.1.4.1 1999/05/06 15:33:19 perja Exp $
17   */
18  public class Reservation implements Serializable
19  {
20      private Person myPerson = null;
21      private Room myRoom    = null;
22      private TimeInterval myTimeInt = null;
23      private int myReservationNumber = 0;
24
25      public Reservation( Person person, Room room, TimeInterval timeInt )
26      {
27          myPerson = person;
28          myRoom  = room;
29          myTimeInt = timeInt;
30
```

```
31        // This method should also check for reservations that are in
32        // conflict.
33     }
34
35     public void setReservationNumber( int number )
36     {
37        myReservationNumber = number;
38     }
39
40     public int getReservationNumber()
41     {
42        return myReservationNumber;
43     }
44
45     public Person getPerson()
46     {
47        return myPerson;
48     }
49
50     public Room getRoom()
51     {
52        return myRoom;
53     }
54
55     public String getStartTime()
56     {
57        return myTimeInt.getStartTime();
58     }
59
60     public TimeInterval getTimeInterval()
61     {
62        return myTimeInt;
63     }
64 }
```

A.24 resource/tools/ReservationList.java

```
1  package morp.resource.tools;
2
3  // java imports
4  import java.io.Serializable;
5  import java.util.Hashtable;
6  import java.util.Enumeration;
7  import java.util.Vector;
8  import java.util.Calendar;
9  import java.util.Date;
10 import java.util.TimeZone;
11
12 // morp.resource.tools imports
13 import morp.resource.repository.Person;
14 import morp.resource.repository.Room;
15
16 /**
17  * ReservationList - contains all reservations and methods to add,
18  * remove, find etc. reservations.
19  *
20  * @author  Per Thomas Jahr <perja@ifi.uio.no> 5.3.1999
21  * @version
22  * $Id: ReservationList.java,v 1.1.1.1.4.4 1999/06/04 13:38:42 perja Exp $
23  */
24 public class ReservationList implements IReservationList, Serializable
```

```
{
   private static ReservationList myInstance = null; // singleton
   private static Hashtable myReservationList = null;

   private ReservationList()
   {
      myReservationList = new Hashtable();
   }

   // singleton
   public static ReservationList getInstance()
   {
      if ( myInstance == null )
      {
         myInstance = new ReservationList();
      }
      return myInstance;
   }

   /**
    * This method creates a new reservation object and stores it in
    * the reservationlist
    */
   public void add( Person person, Room room, int year, int month, int date,
                int hour, int minute, int duration )
   {
      // creating a timeinterval object
      Calendar cal = Calendar.getInstance();
      cal.set( year, month , date, hour, minute );
      Date startDate = cal.getTime();

      // add the duration in hours
      cal.add( Calendar.HOUR, duration );
      Date endDate = cal.getTime();
      TimeInterval timeInt = null;

      try
      {
         timeInt = new TimeInterval( startDate, endDate );
      }
      catch ( TimeIntervalException e1 )
      {
         System.err.println( e1.getMessage() );
      }

      // creating a new reservation object
      Reservation newReservation = new Reservation( person, room, timeInt );

      // set the id of this object
      // How should reservations be identified? (id)
      // using size of hashtable (number of elements in it) as identifier
      if ( myReservationList == null )
         System.out.println( "ReservationList: myReservationList == null");

      int id = myReservationList.size() + 1;
      newReservation.setReservationNumber( id );
      myReservationList.put( new Integer( id ), newReservation );

      System.out.println( "Reservation added to list: "+
                     newReservation.getPerson().getName() + " "
                     + newReservation.getRoom().getName() + " "
                     + newReservation.getStartTime() );
```

```
87
88          System.out.println( "(ReservationList) Length of reservation list: "
89                              + myReservationList.size() );
90       }
91
92       public void remove( int reservationNumber )
93       {
94          myReservationList.remove( new Integer( reservationNumber) );
95       }
96
97       public Reservation find( int reservationNumber )
98       {
99          return (Reservation)myReservationList.get( new Integer( reservationNumber) );
100      }
101
102      public boolean isPresent( int reservationNumber )
103      {
104         return myReservationList.containsKey( new Integer( reservationNumber) );
105      }
106
107      public Reservation[] getReservations()
108      {
109         Enumeration resEnum = myReservationList.elements();
110         int size = myReservationList.size();
111         Reservation resResult[] = new Reservation[size];
112
113         for ( int i = 0; i < size; i++ )
114         {
115            resResult[i] = (Reservation)resEnum.nextElement();
116         }
117
118         return resResult;
119      }
120
121      public Reservation[] getReservationsForRoom( Room room )
122      {
123         Vector reservationVector = null;
124         Reservation resultList[] = null;
125         Reservation tmpReservation = null;
126         Enumeration reservationEnum = null;
127
128         reservationEnum = myReservationList.elements();
129         reservationVector = new Vector();
130
131         if ( !myReservationList.isEmpty() )
132         {
133            while ( reservationEnum.hasMoreElements() )
134            {
135               tmpReservation = (Reservation)reservationEnum.nextElement();
136
137               if ( tmpReservation.getRoom().equals( room ) )
138               {
139                  reservationVector.addElement( tmpReservation );
140               }
141            }
142
143            reservationVector.copyInto( resultList =
144                                    new Reservation[reservationVector.size()] );
145         }
146         else
147            resultList = null;
148
```

```
149       return resultList;
150    }
151
152    public Reservation[] getReservationsForPerson( Person person )
153    {
154       System.out.println( "ReservationList, getResForPer: inside");
155       Vector reservationVector = null;
156       Reservation resultList[] = null;
157       Reservation tmpReservation = null;
158       Enumeration reservationEnum = null;
159
160       reservationVector = new Vector();
161       System.out.println( "ReservationList: myReservationList:"
162                      + myReservationList );
163       reservationEnum = myReservationList.elements();
164       if ( !myReservationList.isEmpty() )
165       {
166          while ( reservationEnum.hasMoreElements() )
167          {
168             tmpReservation = (Reservation)reservationEnum.nextElement();
169
170             System.out.println( "ReservationList getPerson"
171                            + tmpReservation.getPerson() );
172             System.out.println( "ReservationList person (parameter): "+ person );
173
174             if ( tmpReservation.getPerson().getName().equals( person.getName() ) )
175             {
176                System.out.println( "ReservationList equals! ");
177                reservationVector.addElement( tmpReservation );
178             }
179          }
180          reservationVector.copyInto( resultList =
181                                new Reservation[reservationVector.size()] );
182       }
183       else
184       {
185          resultList = null;
186          System.out.println( "ReservationList, getResForPer: result == null");
187       }
188
189       System.out.println( "ReservationList, getResForPer: resultList.length = "
190                      + resultList.length );
191
192       return resultList;
193    }
194 }
```

A.25 resource/tools/TimeInterval.java

```
1  package morp.resource.tools;
2
3  // java imports
4  import java.io.Serializable;
5  import java.util.Calendar;
6  import java.util.Date;
7
8  /**
9   * TimeInterval represents the start and end time of a reservation.
10  *
11  * @author  Per Thomas Jahr <perja@ifi.uio.no> 8. 3.1999
12  * @version $Id: TimeInterval.java,v 1.1.1.1.4.1 1999/05/06 15:33:19 perja Exp $
```

```
13  */
14 public class TimeInterval implements Serializable
15 {
16    private Date myStartDate = null;
17    private Date myEndDate = null;
18
19    public TimeInterval( Date start, Date end ) throws TimeIntervalException
20    {
21       // Check that start comes before end
22       if ( !start.before( end ) )
23       {
24          throw new TimeIntervalException();
25       }
26
27       myStartDate = start;
28       myEndDate = end;
29    }
30
31    public String getStartTime()
32    {
33       Calendar cal = Calendar.getInstance();
34       cal.setTime( myStartDate );
35
36       int year = cal.get( Calendar.YEAR );
37       int month = cal.get( Calendar.MONTH );
38       int date = cal.get( Calendar.DATE );
39       int hour = cal.get( Calendar.HOUR_OF_DAY );
40
41       String startTime = ( date + "."+ month + "."+ year + ":"+ hour );
42
43       return startTime;
44    }
45
46    /**
47     * Check if two timeintervals are in any conflict (overlapping)
48     */
49    boolean hasTimeConflictWith( TimeInterval otherTimeInterval )
50    {
51       boolean check1 = false;
52       boolean check2 = false;
53
54       Date xStart = myStartDate;
55       Date xEnd = myEndDate;
56       Date yStart = otherTimeInterval.getStartDate();
57       Date yEnd = otherTimeInterval.getEndDate();
58
59       check1 = xStart.before( yStart ) &&
60          xEnd.before( yEnd ) &&
61          xEnd.before( yStart );
62
63       check2 = yStart.after( xStart ) &&
64          yEnd.after( xEnd ) &&
65          yStart.after( xEnd );
66
67       // one of the two checks must be true
68       if ( ( check1 || check2 ) )
69       {
70          // No conflict
71          return false;
72       }
73       else
74          // Conflict
```

```
75          return true;
76    }
77
78    Date getStartDate()
79    {
80       return myStartDate;
81    }
82
83    Date getEndDate()
84    {
85       return myEndDate;
86    }
87 }
```

A.26 resource/tools/TimeIntervalException.java

```
1  package morp.resource.tools;
2
3  /**
4   * Exception for timeinterval
5   *
6   * @author  Per Thomas Jahr <perja@ifi.uio.no> 8. 3.1999
7   * @version
8   * $Id: TimeIntervalException.java,v 1.1.1.1 1999/04/23 07:37:48 perja Exp $
9   */
10 public class TimeIntervalException extends Exception
11 {
12    TimeIntervalException()
13    {
14    }
15 }
```

Tillegg B

Oversikt over grensesnitt

Dette vedlegget inneholder en oversikt over grensesnittene i MORP.

B.1 IMorp.java

```
1  package morp;
2
3  // morp imports
4  import morp.resource.repository.Person;
5  import morp.resource.repository.Room;
6  import morp.resource.tools.ReservationList;
7
8  /**
9   * Interface IMorp is the main interface to lookup organisations,
10  * rooms and persons in the Morp system. All classes that should
11  * support lookup of these resources must implement this interface. This
12  * interface also contains methods to register and unregister clients.
13  *
14  * @author  Per Thomas Jahr <perja@ifi.uio.no> 26. 4.1999
15  * @version $Revision: 1.1.2.5 $
16  */
17 public interface IMorp
18 {
19    /**
20     * Gets all organisations from the server.
21     *
22     * @return an array of Organsiations
23     */
24    public Organisation[] getOrganisations();
25
26    /**
27     * Gets a organisation object.
28     *
29     * @param name  the name of the organisation
30     * @return a organisation if it exists (otherwise null)
31     */
32    public Organisation getOrganisation( String name );
33
34    /**
35     * Gets all organisation names.
36     *
37     * @return an array of Strings
```

```
38      */
39     public String[] getOrganisationNames();
40
41     /**
42      * Gets the Person object from the persons name. This method will
43      * search all the organisations.
44      *
45      * @param personName the name of the person
46      * @return a Person object if it exists (otherwise null)
47      */
48     public Person findPerson( String personName );
49
50     /**
51      * Gets the Room object from the room name. This method will
52      * search all the organisations.
53      *
54      * @param roomName the name of the room
55      * @return a Room object if it exists (otherwise null)
56      */
57     public Room findRoom( String roomName );
58
59     /**
60      * Registers the clients name and port at the server. Use this
61      * method to notify the server at startup.
62      *
63      * @param name the name of the client to register
64      * @param port the port number of the client to register */
65     public void registerClient( String name, String port );
66
67     /**
68      * Unregister the clients name and port at the server.
69      *
70      * @param name the name of the client to unregister
71      * @param port the port of the client to unregister
72      * @return true if the signoff was successful, false otherwise
73      */
74     public boolean signoffClient( String name, String port );
75 }
```

B.2 IOrganisation.java

```
1  package morp;
2
3  // morp imports
4  import morp.resource.repository.Person;
5  import morp.resource.repository.Room;
6
7  /**
8   * Interface IOrganisation is the interface to access rooms and
9   * persons of a organisation. All classes that should support these
10  * methods must implement this interface.
11  *
12  * @author  Per Thomas Jahr <perja@ifi.uio.no> 5. 5.1999
13  * @version $Revision: 1.1.2.1 $ */
14 public interface IOrganisation
15 {
16     /**
17      * Gets the name of this organisation.
18      *
19      * @return the name of the organisation
20      */
```

```
21    public String getName();
22
23    /**
24     * Gets all the rooms for this organisation.
25     *
26     * @return an array of Room objects
27     */
28    public Room[] getRooms();
29
30    /**
31     * Gets all room names for this organisation.
32     *
33     * @return an array of Strings representing the names
34     */
35    public String[] getRoomNames();
36
37    /**
38     * Gets all the persons for this organisation.
39     *
40     * @return an array of Strings representing the names
41     */
42    public Person[] getPersons();
43
44    /**
45     * Finds a person object from the name of the person.
46     *
47     * @param personName the name of the person
48     * @return the person object
49     */
50    public Person findPerson( String personName );
51
52    /**
53     * Finds a room object from the name of the room.
54     *
55     * @param roomName the name of the room
56     * @return the room object
57     */
58    public Room findRoom( String roomName );
59
60    /**
61     * Gets a list of person names for this organisation
62     *
63     * @return an array of Strings representing the names
64     */
65    public String[] getPersonNames();
66 }
```

B.3 agent/IUpgradeAgent.java

```
1  package morp.agent;
2
3  /**
4   * IUpgradeAgent - the agent interface
5   *
6   * @author  Per Thomas Jahr <perja@ifi.uio.no> 19. 5.1999
7   * @version $Revision: 1.1.2.4 $
8   */
9  public interface IUpgradeAgent
10 {
11    public void go();
12    public String[] getResults();
```

```
13    public void createTour( String urls[] );
14 }
```

B.4 resource/tools/IAgendaList.java

```
1  package morp.resource.tools;
2
3  /**
4   * IAgendaList - this is an interface for the AgendaList class
5   *
6   * @author  Per Thomas Jahr <perja@ifi.uio.no> 11. 8.1999
7   * @version $Id: IAgendaList.java,v 1.1.2.1 1999/08/11 18:03:13 perja Exp $
8   */
9  public interface IAgendaList
10 {
11    public void addAgenda( Reservation reservation, Agenda agenda );
12    public Agenda getAgenda( Reservation reservation );
13 }
```

B.5 resource/tools/IReservationList.java

```
1  package morp.resource.tools;
2
3  // morp imports
4  import morp.resource.repository.Person;
5  import morp.resource.repository.Room;
6
7  /**
8   * Interface
9   *
10  * @author  Per Thomas Jahr <perja@ifi.uio.no> 6. 5.1999
11  * @version $Revision: 1.1.2.1 $
12  */
13 public interface IReservationList
14 {
15    public void add( Person person, Room room, int year, int month, int date,
16                     int hour, int minute, int duration );
17    public void remove( int reservationNumber );
18    public Reservation find( int reservationNumber );
19    public boolean isPresent( int reservationNumber );
20    public Reservation[] getReservations();
21    public Reservation[] getReservationsForRoom( Room room );
22    public Reservation[] getReservationsForPerson( Person person );
23 }
```